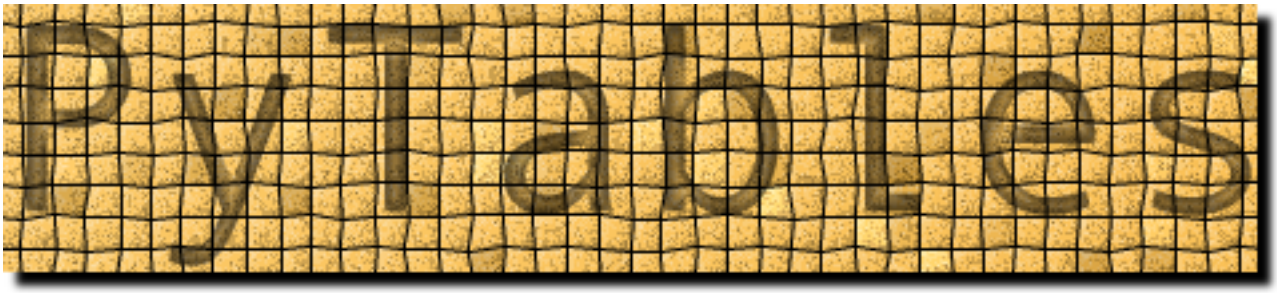
Francesc Altet • Ivan Vilata • Scott Prater • Vicent Mas • Tom Hedley • Antonio Valentino • Jeffrey Whitaker

PyTables User's Guide



Hierarchical datasets in Python - Release 1.3

Altet, Francesc:

PyTables User's Guide

Hierarchical datasets in Python - Release 1.3

Typeset by Francesc Altet, Scott Prater, Ivan Vilata, Vicent Mas, Tom Hedley, Antonio Valentino and Jeffrey Whitaker
Day of print: $LastChangedDate: 2006-03-30 14:05:30 +0200 (Thu, 30 Mar 2006) $

Contents

Part I

The `PyTables` Core Library

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).

—A wise Catalan
in "Cien años de soledad"
Gabriel García Márquez

Chapter 1

Introduction

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 NCSA).

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 name field and types 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    = IntCol()        # integer (equivalent to Int32Col)
    class Properties(IsDescription):  # A sub-structure (nested data-type)
        pressure = Float32Col(shape=(2,3)) # 2-D float array (single-precision)
        energy   = FloatCol(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 the **array** objects that 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 in any 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 HDF5 powerful data management features and numarray flexibility and high-performance manipulation of large sets of objects organized in grid-like fashion to provide these features:

- *Support for table entities:* You can tailor your data adding or deleting records in your tables. A large number of rows (up to 2**62), i.e. much more than will fit into memory is supported as well.
- *Multidimensional and nested table cells:* You can declare a column to consist of general array cells as well as scalars, which is the only dimensionality allowed the majority of relational databases. You can even declare columns that are made of other columns (of different types), which is known as struct types.
- *Indexing support for columns of tables:* Very useful if you have large tables and you want to quickly look up for values in columns satisfying some criteria.
- *Support for numerical arrays:* NumPy (see Oliphant *et al.*), Numeric (see Ascher *et al.*) and numarray (see Greenfield *et al.*) 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 can access to only 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 be variable 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 the data. PyTables builds up an *object tree* in memory that replicates the underlying file data structure. Access to the file objects is achieved by walking through and manipulating this object tree.
- *User defined metadata:* Besides supporting system metadata (like the number of rows of a table, shape, flavor, etc.) the user may specify its own metadata (as for example, room temperature, or protocol for IP traffic that was collected) that complement the meaning of his 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 into a special UnImplemented class (see 4.14), that will let the user see that the data is there, although it would 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 has HDF5 itself) with little-endian/big-endian byte orderings 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. 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 the element's 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. Only the *metadata* (i.e. special data that describes the structure of the actual data) is loaded. The actual data is not read until you 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, name columns, data types in columns, numbers of rows, or, in the case of arrays, the shapes, typecodes, etc. of the array. 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 to note that, from version 1.2 on, PyTables sports a *node cache system* that loads nodes on demand, and unloads nodes that have not been used for some time (i.e. following a *Least Recent Used* schema). This feature allows opening HDF5 files with large hierarchies very quickly and with a low memory consumption, while retaining all the powerful browsing capabilities of the previous implementation of the object tree. See Altet and Vilata for more facts about the advantages introduced by this new node cache system.

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

```
from tables import *

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

# Open a file in "w"rite mode
fileh = openFile("objecttree.h5", mode = "w")
# Get the HDF5 root group
root = fileh.root

# Create the groups:
```

[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 Mertz)

```
group1 = fileh.createGroup(root, "group1")
group2 = fileh.createGroup(root, "group2")

# Now, create an array in the root group
array1 = fileh.createArray(root, "array1",
                           ["this is", "a string array"], "String array")
# Create 2 new tables in group1 and group2
table1 = fileh.createTable(group1, "table1", Particle)
table2 = fileh.createTable("/group2", "table2", Particle)
# Create one more Array in group1
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, allowing you to work with it in exactly the same way as when you originally created it.

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 is always created when reading any supported generic HDF5 file). It's worthwhile to take your time to understand it[4]. It will help you to avoid programming mistakes.

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

[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.

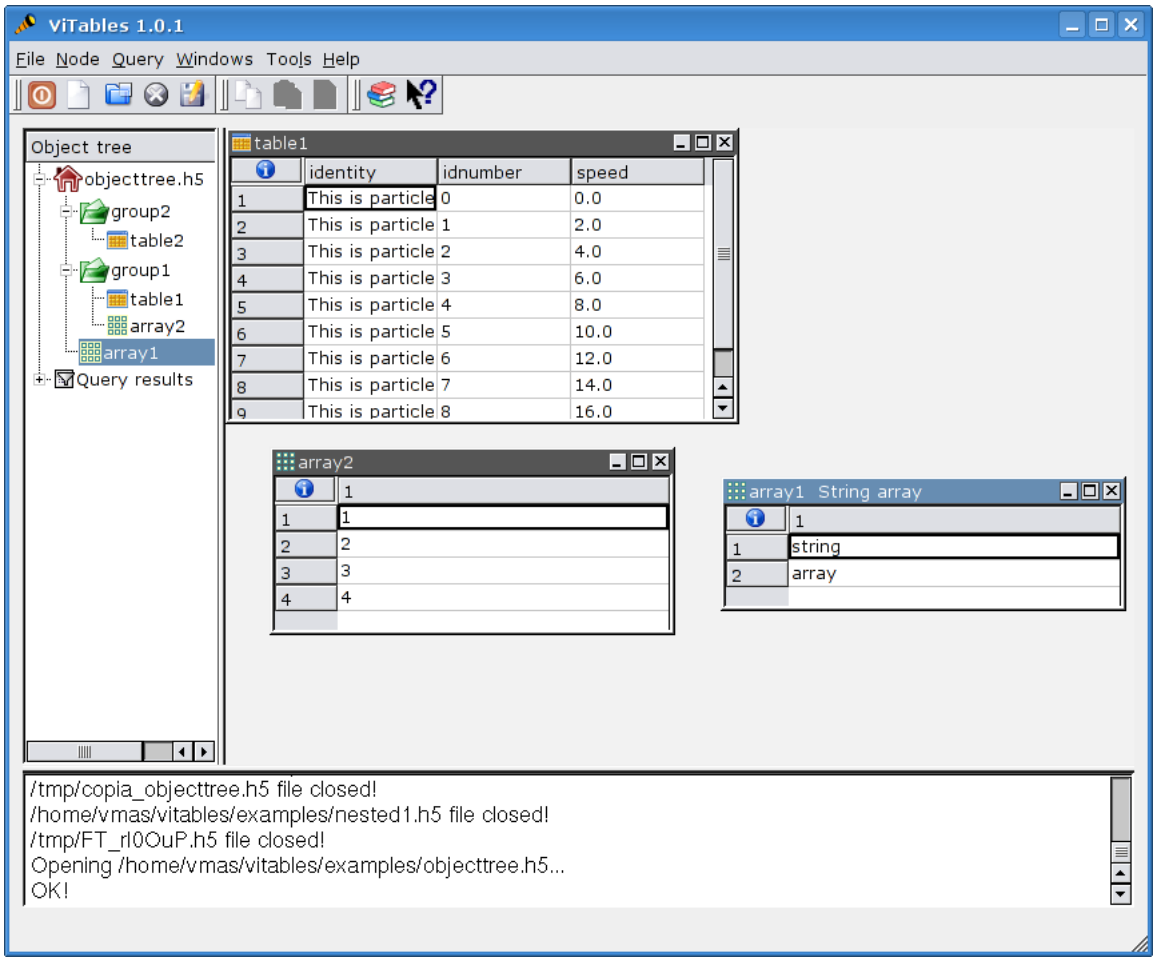


Figure 1.1: An HDF5 example with 2 subgroups, 2 tables and 1 array.

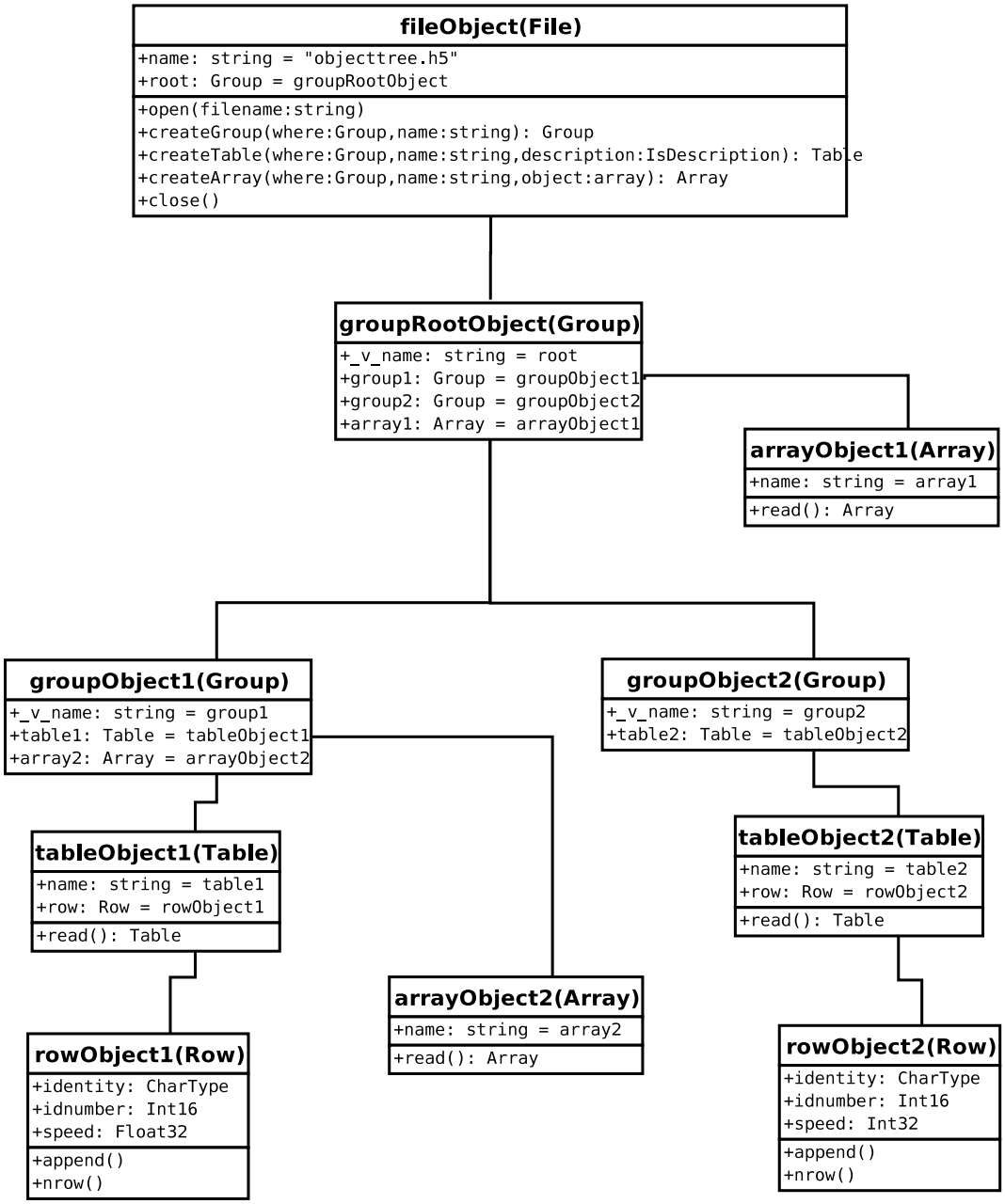


Figure 1.2: A `PyTables` object tree example.

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

—Albert Einstein

Chapter 2

Installation

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 go to the next section. But if you are running Windows and want to install precompiled binaries jump to section 2.2). In addition, packages are available for many different Linux distributions, for instance T2 Project, `RockLinux`, `Debian`, or `Gentoo`, among others. There also packages for other Unices like `FreeBSD` or `MacOSX`

2.1 Installation from source

These instructions are for both Unix/Linux and Windows systems. If you are using Windows, it is assumed that you have a recent version of `MS Visual C++` (>= 6.0) 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 Ewing) and 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 distribution.

To compile `PyTables` you will need a recent version of `Python`, the `HDF5` (C flavor) library, and the `numarray` (see Greenfield *et al.*) package. Although you won't need `NumPy` (see Oliphant *et al.*) or `Numeric` (see Ascher *et al.*) in order to compile PyTables, they are supported; you only need a reasonably recent version of them (>= 0.9.6 for NumPy and >= 24.2 for Numeric) if you plan on using them in your applications. If you already have `NumPy` and/or `Numeric` installed, the test driver module will detect them and will run the tests for `NumPy` and/or `Numeric` automatically.

2.1.1 Prerequisites

First, make sure that you have at least Python 2.3 or 2.4 (Python 2.2 is unsupported), HDF5 1.6.4 and numarray 1.5.0 or higher installed (I'm using HDF5 1.6.5 and numarray 1.5.1 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 Gailly and Adler), which is also required by `HDF5` as well. You may also optionally install the excellent `LZO` compression library (see Oberhumer and section 5.3). The high-performance bzip2 compression library can also be used with PyTables (see Seward). The use of the UCL compression library is in process of being *deprecated*[1], so it is recommended to not use it unless you have to (you still have data files compressed with UCL). Meanwhile, you can force its support in PyTables by passing the `--force-ucl` flag to `setup.py` (see later).

Unix `setup.py` will detect `HDF5`, `LZO`, `UCL` or `bzip2` libraries and include files under `/usr` or `/usr/local`; this will cover most manual installations as well as installations from packages. If

[1] This is because of recurrent memory problems in some platforms (perhaps some bad interaction between UCL and *something* else). Eventually, UCL support will be dropped in the future, so, please, refrain to create datasets compressed with it.

setup.py can not find libhdf5 (or liblzo, libucl 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, setting the values of the HDF5_DIR, LZO_DIR, UCL_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/hdf5-1.6.5
--lzo=/stuff/lzo-1.08
--bzip2=/stuff/bzip2-1.0.3
```

You can force the compilation of the deprecated UCL compressor by passing the --force-ucl flag:

```
--ucl=/stuff/ucl-1.03 --force-ucl
```

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/hdf5-1.6.5/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="hdf5-1.6.5"
  LIBS="hdf5-1.6.5 nsl"
```

Windows Once you have installed the prerequisites, setup.py needs to know where the necessary library *stub* (.lib) and *header* (.h) files are installed. Set the following environment variables:

HDF5_DIR Points to the root HDF5 directory (where the include/ and dll/ directories can be found). *Mandatory*.

LZO_DIR Points to the root LZO directory (where the include/ and lib/ directories can be found). *Optional*.

BZIP2_DIR Points to the root bzip2 directory (where the include/ and lib/ directories can be found). *Optional*.

UCL_DIR Points to the root UCL directory (where the include/ and lib/ directories can be found). *Optional, but discouraged*.

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
```

Or, you can pass this information to setup.py by setting the appropriate arguments on the command line. For example:

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

You can force the compilation of the deprecated UCL compressor by passing the --force-ucl flag:

```
--ucl=c:\stuff\ucl-1-02 --force-ucl
```

You can get ready-to-use Windows binaries and other development files for most of those libraries from the GnuWin32 project (see Wilke *et al.*).

2.1.2 `PyTables` package installation

Once you have installed the HDF5 library and the numarray 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, change into the `tables/tests` directory and execute this command:

   Unix In the shell `sh` and its variants:

   ```
   PYTHONPATH=../..  python test_all.py
   ```

   Windows Open a DOS terminal and type:

   ```
   set PYTHONPATH=..\..
   python test_all.py
   ```

   If you would like to see verbose output from the tests simply add the flag `-v` and/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 `types` test:

   ```
   python test_types.py -v
   ```

   If a test fails, please enable verbose output (the `-v` flag **and** `verbose` option), run the failing test module again, and, very important, get your `PyTables` version information by running the command:

   ```
   python test_all.py --show-versions
   ```

   and send back the output 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 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 or in `python_installation_path\Lib\site-packages\tables` (the last directory may have not exist yet, so if you want to install the DLLs there, you should do so *after* installing the PyTables package). 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 2.3, 2.4 or higher (Python 2.2 is unsupported), HDF5 1.6.5 or higher and numarray 1.5.0 or higher installed (I have built the PyTables binaries using HDF5 1.6.5 and numarray 1.5.1).

For the HDF5 library it should be enough to manually copy the hdf5dll.dll, zlib1.dll **and** szipdll.dll files to a directory in your PATH environment variable (for example C:\WINDOWS\SYSTEM32) or
python_installation_path\Lib\site-packages\tables (the last directory may have not exist yet, so if you want to install the DLLs there, you should do so *after* installing the PyTables package).

Caveat: When downloading the binary distribution for HDF5 libraries, select one compiled with MSVC 6.0 if you are using Python 2.3.x, such as the package 5-165-win.zip. The file 5-165-win-net.zip was compiled with the MSVC 7.1 (aka ".NET 2003") and you **must** choose if you want to run PyTables with Python 2.4.x series. You have been warned!

To enable compression with optional LZO or bzip2 libraries (see the section 5.3 for hints about how they may be used to improve performance), fetch and install the LZO (choose v1.x, LZO v2.x is not supported in precompiled Windows builds) and bzip2 binaries from Wilke *et al.*[2]. Normally, you will only need to fetch and install the
<package>-<version>-bin.zip file and copy the lzo1.dll or bzip2.dll files in a directory in the PATH environment variable, or in python_installation_path\Lib\site-packages\tables (the last directory may have not exist yet, so if you want to install the DLLs there, you should do so *after* installing the PyTables package), so that they 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 any of LZO or bzip2 dynamic libraries if you don't want to.

[2] Note that support for the UCL compressor has been declared deprecated and has not been added in the binary build of PyTables for Windows.

2.2.2 `PyTables` package installation

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

You can (*you should*) test your installation by unpacking the source tar-ball, changing to the `tables/tests/` subdirectory and executing the `test_all.py` script. If all the tests pass (possibly with a few warnings, related to the potential unavailability of LZO or bzip2 libs) you already have a working, well-tested copy of `PyTables` installed! If any test fails, please try to locate which test module is failing and execute:

```
python test_<module>.py -v verbose
```

and also:

```
python test_all.py --show-versions
```

and mail the output 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`.

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!

—M'aclame a tu
Lyrics: Vicent Andrés i Estellés
Music: Ovidi Montllor

Chapter 3

Tutorials

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 `numarray` 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 very reduced set of first-level primitives, so you may consider using the alternative:

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

which will export in your caller application namespace the following functions: `openFile()`, `copyFile()`, `isHDF5File()`, `isPyTablesFile()` and `whichLibVersion()`. This is a rather reduced set of functions, and for convenience, we will use this technique to access them.

If you are going to work with `numarray` (or `NumPy` 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 numarray      # or "import numpy" 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:

```
>>> 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    = IntCol()        # integer (equivalent to Int32Col)
...     pressure  = Float32Col()    # float  (single-precision)
...     energy    = FloatCol()      # 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 4.16.2 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` (see 4.1.2) function to create a `PyTables` file:

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

`openFile` (see 4.1.2) 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 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 4.2.2) 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 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 4.6) object as a branch off the newly-created group. We do that by calling the `createTable` (see 4.2.2) 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
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:00:13 2003'
/ (Group) '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', trMap={}, rootUEP='/')
/ (Group) 'Test file'
/detector (Group) 'Detector information'
/detector/readout (Table(0,)) 'Readout example'
  description := {
    "ADCcount": Col('UInt16', shape=1, itemsize=2, dflt=0),
    "TDCcount": Col('UInt8', shape=1, itemsize= 1, dflt=0),
    "energy": Col('Float64', shape=1, itemsize=8, dflt=0.0),
    "grid_i": Col('Int32', shape=1, itemsize=4, dflt=0),
    "grid_j": Col('Int32', shape=1, itemsize=4, dflt=0),
    "idnumber": Col('Int64', shape=1, itemsize=8, dflt=0),
    "name": Col('CharType', shape=1, itemsize=16, dflt=None),
    "pressure": Col('Float32', shape=1, itemsize=4, dflt=0.0) }
  byteorder := little
```

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 4.6.4) 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)
...     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 4.6.4). 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()
```

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 4.6.2). The iterator returns values until all the data in table is exhausted. These rows are filtered using the expression:

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

We select the value 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.

That's enough about selections. 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'
  type = Float64
  itemsize = 8
  flavor = 'numarray'
  byteorder = 'little'
```

We already know the first two parameters of the `createArray` (see 4.2.2) 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 `numarray` 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(4,) 'Name column selection'
  type = 'CharType'
  itemsize = 16
  flavor = 'List'
  byteorder = 'little'
```

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 4.2.2) 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
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:00:13 2003'
/ (Group) '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 outputs as displayed by the "ptdump" `PyTables` utility (located in `utils/` directory):

```
$ ptdump tutorial1.h5
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:40:51 2003'
/ (Group) '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'
```

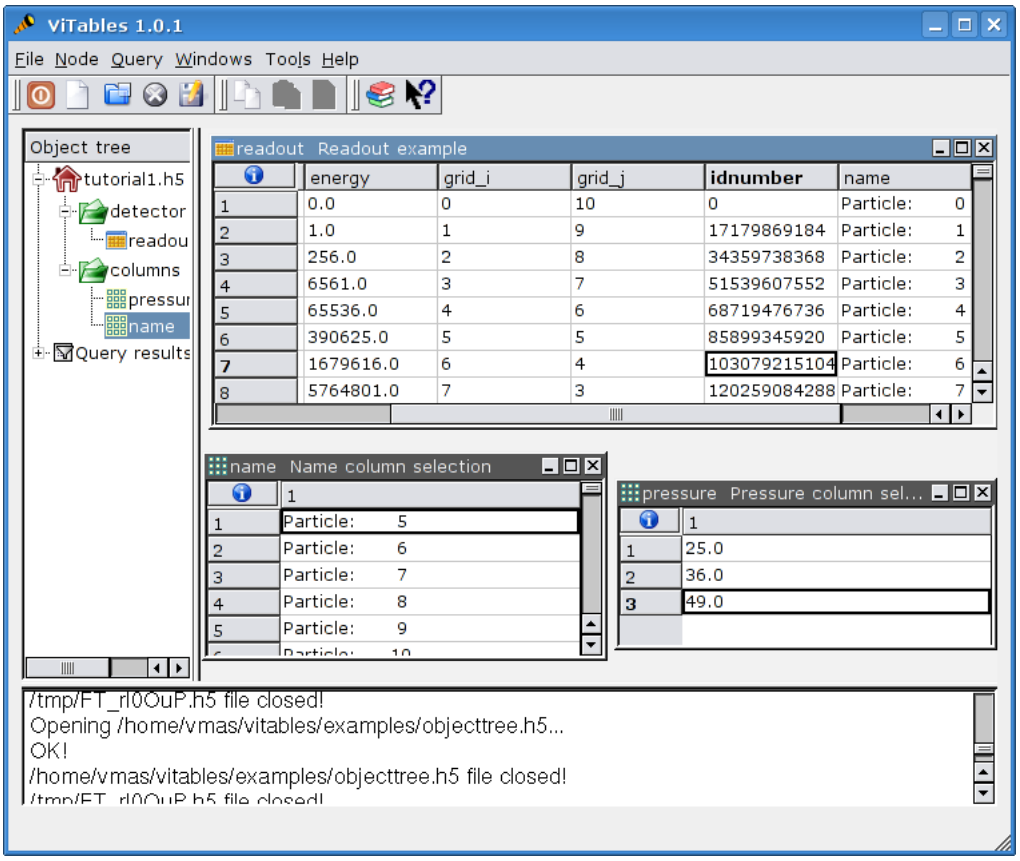


Figure 3.1: The initial version of the data file for tutorial 1, with a view of the data objects.

```
/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 graphical interface .

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
Filename: 'tutorial1.h5' Title: 'Test file' Last modif.: 'Sun Jul 27 14:40:51 2003'
/ (Group) '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 to list all the nodes in the object tree:

```
>>> for node in h5file:
...   print node
...
/ (Group) '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 4.2.2) of the File class to list only the *groups* on tree:

```
>>> for group in h5file.walkGroups("/"):
...   print group
...
/ (Group) '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 4.2.2) 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 4.2.2) 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 4.2.2). 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 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, recursive)` method (4.4.2), 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 4.15). 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 4.15 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 (AttributeSet), 2 attributes:
   [temp_scale := 'Celsius',
    temperature := 18.399999999999999]
```

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_NAME', 'FIELD_1_NAME', 'FIELD_2_NAME', 'FIELD_3_NAME',
 'FIELD_4_NAME', 'FIELD_5_NAME', 'FIELD_6_NAME', 'FIELD_7_NAME', 'NROWS',
 'TITLE', 'VERSION', 'temp_scale', 'temperature']
>>> print table.attrs._f_list("user")
['temp_scale', 'temperature']
>>> print table.attrs._f_list("sys")
['CLASS', 'FIELD_0_NAME', 'FIELD_1_NAME', 'FIELD_2_NAME', 'FIELD_3_NAME',
 'FIELD_4_NAME', 'FIELD_5_NAME', 'FIELD_6_NAME', 'FIELD_7_NAME', 'NROWS',
 'TITLE', 'VERSION']
```

You can also rename attributes:

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

However, you can not set, delete or rename read-only attributes:

```
>>> table.attrs._f_rename("VERSION", "version")
Traceback (most recent call last):
  File ">stdin>", line 1, in ?
  File "/home/falted/PyTables/pytables-0.7/tables/AttributeSet.py",
  line 249, in _f_rename
    raise AttributeError, \
AttributeError: Read-only attribute ('VERSION') cannot be renamed
```

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:  "2.0"
Attribute: TITLE     scalar
    Type:      16-byte null-terminated ASCII string
    Data:  "Readout example"
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: tempScale scalar
    Type:      8-byte null-terminated ASCII string
    Data:  "Celsius"
Attribute: temperature {1}
    Type:      native double
    Data:  18.4
Attribute: NROWS     {1}
    Type:      native int
    Data:  10
Location:  0:1:0:1952
Links:     1
Modified:  2003-07-24 13:59:19 CEST
Chunks:    {2048} 96256 bytes
Storage:   470 logical bytes, 96256 allocated bytes, 0.49% 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.coltypes[name], table.colshapes[name])
...
ADCcount := UInt16, 1
TDCcount := UInt8, 1
energy := Float64, 1
grid_i := Int32, 1
grid_j := Int32, 1
idnumber := Int64, 1
name := CharType, 1
pressure := Float32, 1
```

Here, the `name`, `title`, `nrows`, `colnames`, `coltypes` and `colshapes` attributes (see 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 printing their internal doc strings:

```
>>> print table.__doc__
Represent a table in the object tree.
    It provides methods to create new tables or open existing ones, as
    well as to write/read data to/from table objects over the
    file. A method is also provided to iterate over the rows without
    loading the entire table or column in memory.

    Data can be written or read both as Row instances or numarray
    (NumArray or RecArray) objects or NestedRecArray objects.

    Methods:

        __getitem__(key)
        __iter__()
        __setitem__(key, value)
        append(rows)
        flushRowsToIndex()
        iterrows(start, stop, step)
        itersequence(sequence)
```

```
        modifyRows(start, rows)
        modifyColumn(columns, names, [start] [, stop] [, step])
        modifyColumns(columns, names, [start] [, stop] [, step])
        read([start] [, stop] [, step] [, field [, flavor]])
        reIndex()
        reIndexDirty()
        removeRows(start [, stop])
        removeIndex(column)
        where(condition [, start] [, stop] [, step])
        whereAppend(dstTable, condition [, start] [, stop] [, step])
        getWhereList(condition [, flavor])

    Instance variables:

        description -- the metaobject describing this table
        row -- a reference to the Row object associated with this table
        nrows -- the number of rows in this table
        rowsize -- the size, in bytes, of each row
        cols -- accessor to the columns using a natural name schema
        colnames -- the field names for the table (tuple)
        coltypes -- the type class for the table fields (dictionary)
        colshapes -- the shapes for the table fields (dictionary)
        colindexed -- whether the table fields are indexed (dictionary)
        indexed -- whether or not some field in Table is indexed
        indexprops -- properties of an indexed Table
```

The `help` function is also a handy way to see `PyTables` reference documentation online. Try it yourself with other object docs:

```
>>> help(table.__class__)
>>> 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'
  type = Float64
  itemsize = 8
  flavor = 'numarray'
  byteorder = 'little'
>>> print "  shape: ==>", pressureObject.shape
  shape: ==> (3,)
>>> print "  title: ==>", pressureObject.title
  title: ==> Pressure column selection
>>> print "  type: ==>", pressureObject.type
  type: ==> Float64
```

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 4.10.1 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: <class 'numarray.numarraycore.NumArray'>
>>> nameArray = h5file.root.columns.name.read()
>>> nameArray
['Particle:      5', 'Particle:      6', 'Particle:      7']
>>> 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
>>> pressureObject.name
'pressure'
```

You can see that the `read()` method (see section 4.10.2) returns an authentic `numarray` object for the `pressureObject` instance by looking at the output of the `type()` call. A `read()` of the `nameObject` 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 section 4.2.2 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:

[1] Appending data to arrays is also supported, but you need to create special objects called `EArray` (see 4.12 for more info).

```
>>> 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].

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

[2] Note that you can append not only scalar values to tables, but also fully multidimensional array objects.

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 modif first row of ADCcount-->", table[0]
After modif first row of ADCcount--> (0, 1, 0.0, 0, 10, 0L, 'Particle: 0', 0.0)
>>> table.cols.energy[0] = 2
>>> print "After modif first row of energy-->", table[0]
After modif 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 ADCcount-->", table[0:5]
After modifying slice [2:5] of ADCcount--> RecArray[
(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--> RecArray[
(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),
(2560, 10, 100000000.0, 10, 0, 171798691840L, 'Particle:     10', 100.0),
(2816, 11, 214358881.0, 11, -1, 188978561024L, 'Particle:     11', 121.0),
(3072, 12, 4.0, 12, -2, 206158430208L, 'Particle:     12', 144.0),
(3328, 13, 815730721.0, 13, -3, 223338299392L, 'Particle:     13', 169.0)
]
```

Check that the values has been correctly modified!. **Hint:** remember that column `TDCcount` is the first one, and that `energy` is the third. Look for more info on modifying columns in section 4.9.3.

PyTables also let's 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--> RecArray[
(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 4.6.2 and 4.6.2 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 section 4.6.4) 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(table.cols.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)--> NestedRecArray[
(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 4.10.3). Let's see at how modify data on the `pressureObject` array:

```
>>> 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 section 4.10.3 for more examples on how to use extended slicing in PyTables objects.

Similarly, with and array of strings:

```
>>> print "Before modif-->", nameObject[:]
Before modif--> ['Particle:      5', 'Particle:      6', 'Particle:      7']
>>> nameObject[0] = 'Particle:   None'
```

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

```
>>> 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 4.6.2) deletes the rows in the range (start, stop). It returns the number of rows effectively removed.

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

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

In figure 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`.

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 section 4.2.2 about the different kinds of descriptor objects that can be passed to the `createTable()` method.

```
from numarray import *
from tables import *

# Describe a particle record
class Particle(IsDescription):
    name        = StringCol(length=16) # 16-character String
    lati        = IntCol()             # integer
    longi       = IntCol()             # integer
```

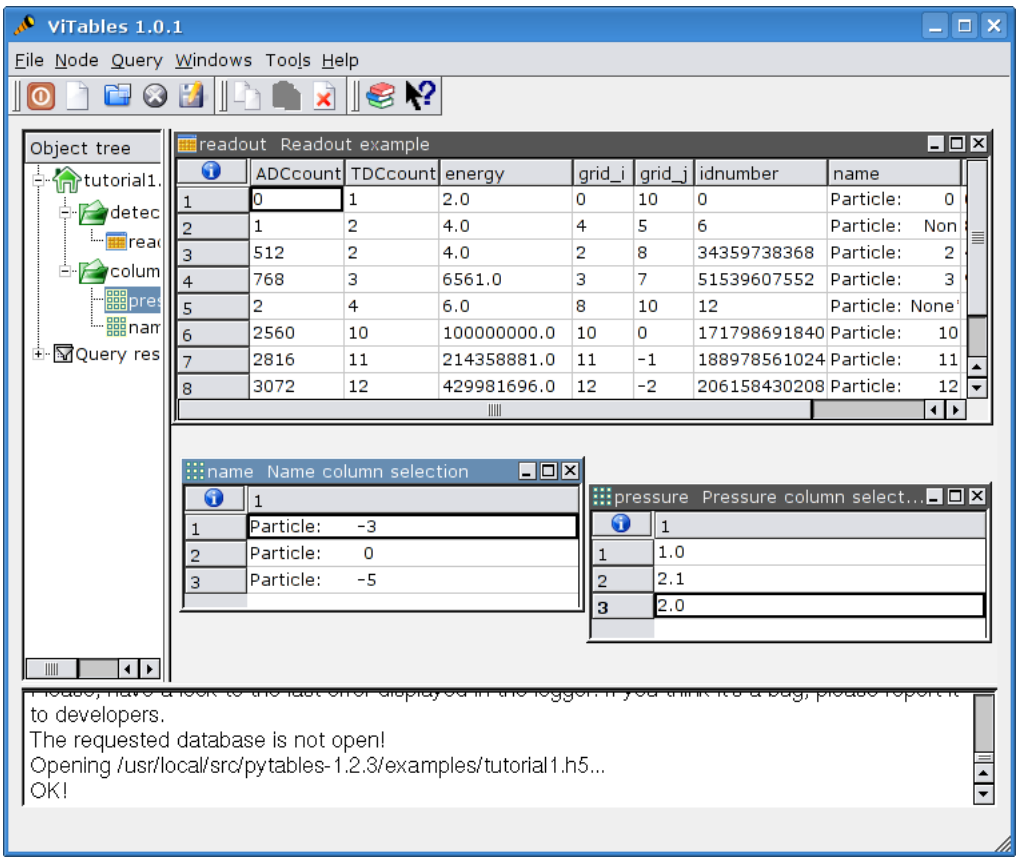


Figure 3.2: The final version of the data file for tutorial 1.

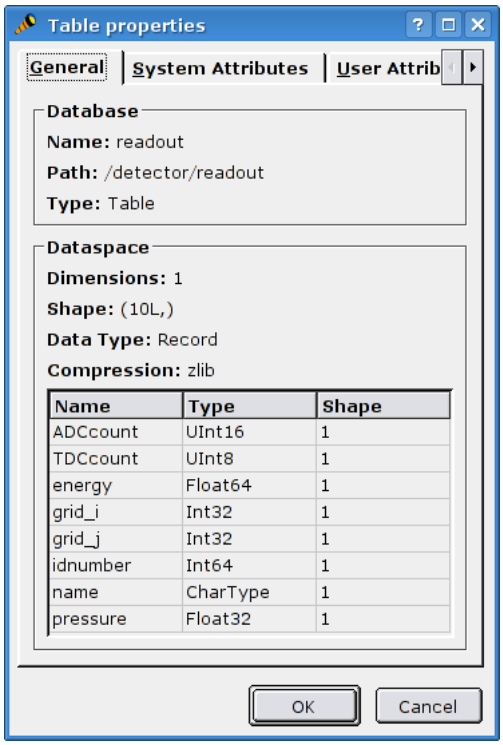


Figure 3.3: General properties of the `/detector/readout` table.

```
    pressure    = Float32Col(shape=(2,3)) # array of floats (single-precision)
    temperature = FloatCol(shape=(2,3))   # array of doubles (double-precision)

# Another way to describe the columns of a table
Event = {
    "name"        : StringCol(length=16),
    "lati"        : IntCol(),
    "longi"       : IntCol(),
    "pressure"    : Float32Col(shape=(2,3)),
    "temperature" : FloatCol(shape=(2,3)),
    }

# 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 the 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 data for 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), shape=(2,4))  # Incorrect
        #particle['pressure'] = array(i*arange(2*3), shape=(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 Events:
for tablename in ("TEvent1", "TEvent2", "TEvent3"):
    # Create a table in the 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 data on 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['xcoord'] = float(i**2)   # Correct spelling
        event['xcoor'] = float(i**2)     # Wrong spelling
        event['ADCcount'] = i * 2        # Correct type
        #event['ADCcount'] = "sss"          # Wrong 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 53, in ?
    particle['pressure'] = array(i*arange(2*3), shape=(2,4))  # Incorrect
  File  "/usr/local/lib/python2.2/site-packages/numarray/numarraycore.py",
 line 281, in array
  a.setshape(shape)
  File "/usr/local/lib/python2.2/site-packages/numarray/generic.py",
 line 530, 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 `numarray` 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 74, in ?
    event['xcoor'] = float(i**2)     # Wrong spelling
  File "src/hdf5Extension.pyx",
 line 1812, in hdf5Extension.Row.__setitem__
    raise KeyError, "Error setting \"%s\" field.\n %s" % \
KeyError: Error setting "xcoor" field.
 Error was: "exceptions.KeyError: 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, in order to test type checking, we will change the next line:

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

to read:

```
    event.ADCcount = "sss"          # Wrong type
```

This modification will cause the following `TypeError` exception to be raised when the script is executed:

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 76, in ?
    event['ADCcount'] = "sss"          # Wrong type
  File "src/hdf5Extension.pyx",
 line 1812, in hdf5Extension.Row.__setitem__
    raise KeyError, "Error setting \"%s\" field.\n %s" % \
KeyError: Error setting "ADCcount" field.
 Error was: "exceptions.TypeError: NA_setFromPythonScalar: bad value type."
```

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`.

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

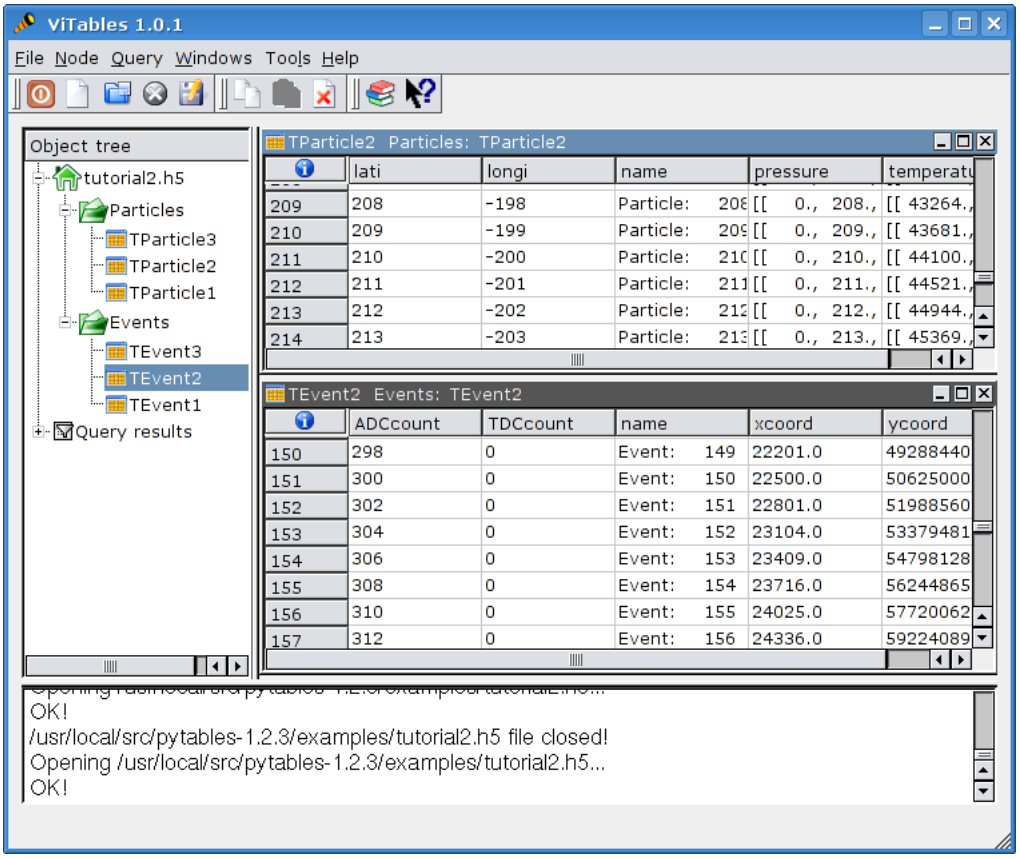


Figure 3.4: Table hierarchy for tutorial 2.

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].

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 page 61) 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 page 61 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 page 62):

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

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

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

```
>>> print fileh
do-undo1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Fri Mar  4 20:22:28 2005'
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
do-undo1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Fri Mar  4 20:22:28 2005'
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 page 62) 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 page 62) to the rescue:

```
>>> fileh.redo()
>>> print fileh
do-undo1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Fri Mar  4 20:22:28 2005'
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
do-undo1.h5 (File) 'Undo/Redo demo 1'
Last modif.: 'Fri Mar  4 20:22:28 2005'
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 page 62).

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

Beginning from version 1.1, PyTables supports the handling of 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 4.17.4) 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)]
>>>
```

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

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 "enum.py", line 222, in __getattr__
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 "enum.py", line 181, in __getitem__
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 "enum.py", line 311, in __call__
ValueError: no enumerated value with that concrete value: 1234
>>>
```

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

3.6.1 Enumerated columns

Columns of an enumerated type can be declared by using the `EnumCol` (see 4.16.2) 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', dtype='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 ?
  File "hdf5Extension.pyx", line 2936, in hdf5Extension.Row.__setitem__
  File "enum.py", line 311, in __call__
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()
>>>
>>> COMMENT("Now print them!")
>>> for r in tbl:
...     ballTime = r['ballTime']
...     ballColor = colors(r['ballColor'])  # notice this
...     print "Ball extracted on %d is of color %s." % (ballTime, ballColor)
...
Ball extracted on 1116501220 is of color white.
Ball extracted on 1116501221 is of color red.
Ball extracted on 1116501222 is of color blue.
Ball extracted on 1116501223 is of color white.
Ball extracted on 1116501224 is of color white.
Ball extracted on 1116501225 is of color green.
Ball extracted on 1116501226 is of color black.
Ball extracted on 1116501227 is of color red.
Ball extracted on 1116501228 is of color white.
Ball extracted on 1116501229 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 4.6.2) to get the enumeration back.

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

3.6.2 Enumerated arrays

`EArray` and `VLArray` leaves can also be declared to store enumerated values by means of the `EnumAtom` (see 4.16.3) 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, shape=(0, 2), flavor='Tuple')
>>> earr = h5f.createEArray('/', 'days', dayRange, title="Working day ranges")
>>>
```

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 4.12.2) 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 ?
  File "enum.py", line 311, in __call__
ValueError: no enumerated value with that concrete value: 1234L
>>>
```

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 and remove the file, and this little tutorial on enumerated types is done:

```
>>> import os
>>> h5f.close()
>>> os.remove('enum.h5')
>>>
```

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):

```
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'])  # An enumerated type

class NestedDescr(IsDescription):
    """A description that has several nested columns"""
    color = EnumCol(colors, 'red', dtype='UInt32', indexed=1) # indexed column
    info1 = Info()
    class info2(IsDescription):
        _v_pos = 1
        name = StringCol(10)
        value = Float64Col(pos=0)
        class info3(IsDescription):
            x = FloatCol(1)
            y = UInt8Col(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.

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

```
>>> 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
10L
>>>
```

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: introducing `NestedRecArray` objects

Now, what happens if we want to read the table? Which data container will be used to keep the data? Well, it's worth trying it:

```
>>> nra = table[::4]
>>> print nra
NestedRecArray[
(((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)
]
>>>
```

We have read one row for each four in the table, giving a result of three rows. What about the container? Well, we can see that it is a new mysterious object known as `NestedRecArray`. If we ask for more info on that:

```
>>> type(nra)
<class 'tables.nestedrecords.NestedRecArray'>
```

we see that it is an instance of the class `NestedRecArray` that lives in the module `nestedrecords` of `tables` package. `NestedRecArray` is actually a subclass of the `RecArray` object of the `records` module of `numarray` package. You can see more info about `NestedRecArray` object in appendix B.

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
13L
>>>
```

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)],
descr=[('info2', [('info3', [('x', '1f8'), ('y', '1u1')]), ('name',
 '1a10'), ('value', '1f8')]), ('info1', [('name', '1a10'), ('value',
 '1f8')]), ('color', '1u4')], shape=3)
```

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 4.7) 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)],
descr=[('info3', [('x', '1f8'), ('y', '1u1')]), ('name', '1a10'),
 ('value', '1f8')],
shape=4)
>>>
```

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)],
descr=[('x', '1f8'), ('y', '1u1')],
shape=4)
>>>
```

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(1,), Description)
  name (Column(1,), CharType)
  value (Column(1,), 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:

```
>>> ycol = table.cols._f_col('info2/info3/y')
>>> ycol
/table.cols.info2.info3.y (Column(1,), UInt8, idx=None)
>>>
```

To sum up, the `cols` accessor is a very handy and powerful way to access data in your nested tables. Be sure 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 4.8) and is useful to discover different meta-information about table data.

Let's see how it looks like:

```
>>> table.description
{
  "info2": {
    "info3": {
      "x": FloatCol(dflt=1, shape=1, itemsize=8, pos=0, indexed=False),
      "y": UInt8Col(dflt=1, shape=1, pos=1, indexed=False)},
    "name": StringCol(length=10, dflt=None, shape=1, pos=1, indexed=False),
    "value": Float64Col(dflt=0.0, shape=1, pos=2, indexed=False)},
  "info1": {
    "name": StringCol(length=10, dflt=None, shape=1, pos=0, indexed=False),
    "value": Float64Col(dflt=0.0, shape=1, pos=1, indexed=False)},
  "color": EnumCol(Enum({'blue': 2, 'green': 1, 'red': 0}), 'red',
 dtype='UInt32', shape=1, pos=2, indexed=1)}
>>>
```

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(length=10, dflt=None, shape=1, pos=0, indexed=False),
    "value": Float64Col(dflt=0.0, shape=1, pos=1, indexed=False)}
>>> table.description.info2.info3
{
      "x": FloatCol(dflt=1, shape=1, itemsize=8, pos=0, indexed=False),
      "y": UInt8Col(dflt=1, shape=1, pos=1, indexed=False)}
>>>
```

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 `NestedRecArrays` objects that imitate the structure of the table (or a subtable!):

```
>>> from tables import nestedrecords
>>> table.description._v_nestedDescr
[('info2', [('info3', [('x', '1f8'), ('y', '1u1')]), ('name', '1a10'),
 ('value', '1f8')]), ('info1', [('name', '1a10'), ('value', '1f8')]),
 ('color', '1u4')]
>>> nestedrecords.array(None, descr=table.description._v_nestedDescr)
array(
[],
descr=[('info2', [('info3', [('x', '1f8'), ('y', '1u1')]), ('name',
 '1a10'), ('value', '1f8')]), ('info1', [('name', '1a10'), ('value',
```

```
 '1f8')]),('color', '1u4')], shape=0)
>>> nestedrecords.array(None, descr=table.description.info2._v_nestedDescr)
array(
[],
descr=[('info3', [('x', '1f8'), ('y', '1u1')]), ('name', '1a10'),
('value', '1f8')], shape=0)
>>>
```

Look the section 4.8 for the complete listing of attributes.

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

```
>>> for coldescr in table.description._v_walk():
...     print "column-->",coldescr
...
column--> Description([('info2', [('info3', [('x', '1f8'), ('y',
 '1u1')]), ('name', '1a10'), ('value', '1f8')]), ('info1', [('name',
 '1a10'), ('value', '1f8')]), ('color', '1u4')])
column--> EnumCol(Enum({'blue': 2, 'green': 1, 'red': 0}), 'red',
 dtype='UInt32', shape=1, pos=2, indexed=1)
column--> Description([('info3', [('x', '1f8'), ('y', '1u1')]),
 ('name', '1a10'), ('value', '1f8')])
column--> StringCol(length=10, dflt=None, shape=1, pos=1, indexed=False)
column--> Float64Col(dflt=0.0, shape=1, pos=2, indexed=False)
column--> Description([('name', '1a10'), ('value', '1f8')])
column--> StringCol(length=10, dflt=None, shape=1, pos=0, indexed=False)
column--> Float64Col(dflt=0.0, shape=1, pos=1, indexed=False)
column--> Description([('x', '1f8'), ('y', '1u1')])
column--> FloatCol(dflt=1, shape=1, itemsize=8, pos=0, indexed=False)
column--> UInt8Col(dflt=1, shape=1, pos=1, indexed=False)
>>>
```

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(13L,)) ''
  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(3L,)) ''
  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. Finally, there exists a function called whichLibVersion that informs about the versions of the underlying C libraries (for example, the HDF5 or the Zlib).

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.

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 4.2.2) 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.

openFile(filename, mode='r', title='', trMap={}, rootUEP="/", filters=None)

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

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

mode The mode to 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+'** is similar to 'a', but the file must already exist.

title If filename is new, this will set a title for the root group in this file. If filename is not new, the title will be read from disk, and this will not have any effect.

trMap A dictionary to map names in the object tree Python namespace into different HDF5 names in file namespace. The keys are the Python names, while the values are the HDF5 names. This is useful when you need to use HDF5 node names with invalid or reserved words in Python.

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. The group has to be named after its HDF5 name and can be a path. If it does not exist, an `HDF5ExtError` exception 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` class (see section 4.17.1) that provides information about the desired I/O filters applicable to the leaves that hang directly from *root* (unless other filters properties are specified for these leaves). Besides, if you do not specify filter properties for its child groups, they will inherit these ones. So, if you open a new file with this parameter set, all the leaves that would be created in the file will recursively inherit this filtering properties (again, if you don't prevent that from happening by specifying other filters on the child groups or leaves).

nodeCacheSize The number of *unreferenced* nodes to be kept in memory. 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.

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`, `ucl` (in process of being *deprecated*) and `bzip2`. If another name is given, a `ValueError` is raised.

4.2 The File class

An instance of this class is returned when a PyTables file is opened with the openFile() 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() 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() 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() 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() 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() method.

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.

trMap A dictionary that maps node names between PyTables and HDF5 domain names. Its initial values are set from the trMap parameter passed to the openFile function. You can change its contents *after* a file is opened and the new map will take effect over any new object added to the tree.

rootUEP The UEP (user entry point) group in the file (see 4.1.2).

filters Default filter properties for the root group (see section 4.17.1).

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

objects A dictionary which maps path names to objects, for every visible node in the tree (deprecated, see note below).

groups A dictionary which maps path names to objects, for every visible group in the tree (deprecated, see note below).

leaves A dictionary which maps path names to objects, for every visible leaf in the tree (deprecated, see note below).

Note: From PyTables 1.2 on, the dictionaries objects, groups and leaves are just instances of objects faking the old functionality. Actually, they internally use File.getNode() (see 4.2.2) and File.walknodes() (see 4.2.2), which are recommended instead.

4.2.2 `File` methods

createGroup(where, name, title='', filters=None)

Create a new Group instance with name *name* in *where* location.

where The parent group where the new group will hang from. *where* parameter can be a path string (for example `"/level1/group5"`), or another Group instance.

name The name of the new group.

title A description for this group.

filters An instance of the `Filters` class (see section 4.17.1) that provides information about the desired I/O filters applicable to the leaves that hangs directly from this new group (unless other filters properties are specified for these leaves). Besides, if you do not specify filter properties for its child groups, they will inherit these ones.

createTable(where, name, description, title='', filters=None, expectedrows=10000)

Create a new `Table` instance with name *name* in *where* location. See the section 4.6 for a description of the `Table` class.

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

name The name of the new table.

description This is an object that describes the table, that is, how many columns has it, and properties for each column: the type, the shape, etc. as well as other table properties.

description can be any of the next several objects:

A user-defined class This should inherit from the `IsDescription` class (see 4.16.1) where table fields are specified.

A dictionary For example, when you do not know beforehand which structure will have your table). See section 3.4 for an example of use.

A `RecArray` This object from the `numarray` package is also accepted, and all the information about columns and other metadata is used as a basis to create the `Table` object. Moreover, if the `RecArray` has actual data this is also injected on the newly created `Table` object.

A `NestedRecArray` Finally, if you want to have nested columns in your table, you can use this object (see appendix B) and all the information about columns and other metadata is used as a basis to create the `Table` object. Moreover, if the `NestedRecArray` has actual data this is also injected on the newly created `Table` object.

title A description for this object.

filters An instance of the `Filters` class (see section 4.17.1) that provides information about the desired I/O filters to be applied during the life of this object.

expectedrows An user estimate of the number of records that will be on table. If not provided, the default value is appropriate for tables until 10 MB in size (more or less). If you plan to save bigger tables you should provide a guess; this will optimize the HDF5 B-Tree creation and management process time and memory used. See section 5.1 for a discussion on that issue.

createArray(where, name, object, title='')

Create a new Array instance with name *name* in *where* location. See the section 4.10 for a description of the Array class.

object The regular array to be saved. Currently accepted values are: NumPy, Numeric, numarray arrays (including CharArray string numarrays) or other native Python types, provided that they 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 zero are not supported (use an EArray object if you want to create an array with one of its dimensions equal to 0).

See createTable description 4.2.2 for more information on the *where*, *name* and *title*, parameters.

createCArray(where, name, shape, atom, title='', filters=None)

Create a new CArray instance with name *name* in *where* location. See the section 4.11 for a description of the CArray class.

shape The *shape* of the objects to be saved.

atom An Atom instance representing the *shape*, *type* and *flavor* of the chunk of the objects to be saved.

See createTable description 4.2.2 for more information on the *where*, *name* and *title*, parameters.

createEArray(where, name, atom, title='', filters=None, expectedrows=1000)

Create a new EArray instance with name *name* in *where* location. See the section 4.12 for a description of the EArray class.

atom An Atom instance representing the *shape*, *type* and *flavor* of the atomic objects to be saved. 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. See section 4.16.3 for the supported set of Atom class descendants.

expectedrows In the case of enlargeable arrays this represents an user estimate about the number of row elements that will be added to the growable dimension in the EArray object. If not provided, the default value is 1000 rows. If you plan to create both much smaller or much bigger EArrays try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used.

See createTable description 4.2.2 for more information on the *where*, *name*, *title*, and *filters* parameters.

createVLArray(where, name, atom=None, title='', filters=None, expectedsizeinMB=1.0)

Create a new VLArray instance with name *name* in *where* location. See the section 4.13 for a description of the VLArray class.

atom An Atom instance representing the shape, type and flavor of the atomic object to be saved. See section 4.16.3 for the supported set of Atom class descendants.

expectedsizeinMB An user estimate about the size (in MB) in the final VLArray object. If not provided, the default value is 1 MB. If you plan to create both much smaller or much bigger VLA's try providing a guess; this will optimize the HDF5 B-Tree creation and management process time and the amount of memory used.

See createTable description 4.2.2 for more information on the *where*, *name*, *title*, and *filters* parameters.

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

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

where can be a `Node` instance 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` 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.

getNodeAttr(where, attrname, name=None)

Returns the attribute *attrname* under *where.name* location.

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

attrname The name of the attribute to get.

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

Sets the attribute *attrname* with value *attrvalue* under *where.name* location. If the node already has a large number of attributes, a `PerformanceWarning` will be issued.

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

attrname The name of the attribute to set on disk.

attrvalue The value of the attribute to set. Any kind of python object (like string, ints, floats, lists, tuples, dicts, small Numeric/NumPy/numarray objects...) can be stored as an attribute. However, if necessary, `(c)Pickle` is automatically used so as to serialize objects that you might want to save (see 4.15 for details).

delNodeAttr(where, attrname, name=None)

Delete the attribute *attrname* in *where.name* location.

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

attrname The name of the attribute to delete on disk.

copyNodeAttrs(where, dstnode, name=None)

Copy the attributes from node *where.name* to *dstnode*.

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

dstnode This is the destination node where the attributes will be copied. It can be either a path string or a `Node` object.

iterNodes(where, classname=None)

Returns an *iterator* yielding children nodes hanging from *where*. These nodes are alpha-numerically sorted by its node name.

where This argument works as in getNode() (see page 58), referencing the node to be acted upon.

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

listNodes(where, classname=None)

Returns a *list* with children nodes hanging from *where*. The list is alpha-numerically sorted by node name.

where This argument works as in getNode() (see page 58), referencing the node to be acted upon.

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

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

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

where, name These arguments work as in getNode() (see page 58), referencing the node to be acted upon.

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

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

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

where, name These arguments work as in getNode() (see page 58), referencing the node to be acted upon.

newparent The destination group that the node will be copied to (a path name or a Group instance). If *newparent* is None, the parent of the source node is selected as the new parent.

newname The name to be assigned to the new copy in its destination (a string). If *newname* is None or not specified, the name of the source node is used.

overwrite Whether the possibly existing node *newparent/newname* should be overwritten or not. Note that trying to copy over an existing node without overwriting it will issue a NodeError.

recursive Specifies whether the copy should recurse into children of the copied node. This argument is ignored for leaf nodes. The default is not recurse.

kwargs Additional keyword arguments may be passed to customize the copying process. The supported arguments depend on the kind of node being copied. The following are some of them:

title The new title for the destination. If None, the original title is used. This only applies to the topmost node for 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 4.17.1). The default is to copy the filter attribute 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 in child leaves 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 in the operation.

renameNode(where, newname, name=None)

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

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

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

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

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

where, name These arguments work as in `getNode()` (see page 58), referencing the node to be acted upon.

newparent The destination group the node will be moved to (a path name or a `Group` instance). If *newparent* is `None`, the original node parent is selected as the new parent.

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

walkGroups(where='/')

Iterator that returns the list of Groups (not Leaves) hanging from (and including) *where*. The *where* Group is listed first (pre-order), then each of its child Groups (following an alpha-numerical order) is also traversed, following the same procedure. If *where* is not supplied, the root object is used.

where The origin group. Can be a path string or `Group` instance.

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

Recursively iterate over the nodes in the `File` instance. It takes two parameters:

where If supplied, the iteration starts from (and includes) this group.

classname *(String)* If supplied, only instances of this class are returned.

Example of use:

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

copyChildren(srcgroup, dstgroup, overwrite=False, recursive=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.

`kwargs` takes keyword arguments used to customize the copying process. See the documentation of `Group._f_copyChildren()` (see 4.4.2) for a description of those arguments.

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 leaves in the object tree.

close()

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

Undo/Redo support

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.

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 mark(), undo(), redo() and other methods to be called.

The filters argument, when specified, must be an instance of class Filters (see section 4.17.1) 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 enableUndo() when the Undo/Redo mechanism is already enabled raises an UndoRedoError.

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 mark(), undo(), redo() and other methods fail with an UndoRedoError.

Calling disableUndo() when the Undo/Redo mechanism is already disabled raises an UndoRedoError.

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 undo() and redo() 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.

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 `enableUndo()`.

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.

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.

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.

4.2.3 `File` special methods

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

`__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__()`

Iterate over the children on the `File` instance. However, this does not accept parameters. This iterator *is recursive*.

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
```

__str__()

Prints a short description of the File object.

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(100L,)) 'This is the table title'
/group0 (Group) ''
/group0/tuple1 (Table(100L,)) 'This is the table title'
/group0/group1 (Group) ''
/group0/group1/tuple2 (Table(100L,)) 'This is the table title'
/group0/group1/group2 (Group) ''
```

__repr__()

Prints a detailed description of the File object.

4.3 The Node class

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. When using a translation map (see 4.2), its *HDF5 name* might differ from its PyTables name.

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 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_file The hosting File instance (see 4.2).

_v_parent The parent Group instance (see 4.4).

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

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

_v_hdf5name The name of this node in the hosting HDF5 file (a string).

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

_v_rootgroup The root group instance. This is deprecated; please use node._v_file.root.

Location independent

_v_objectID The identifier of this node in the hosting HDF5 file.

_v_attrs The associated `AttributeSet` instance (see 4.15).

Attribute shorthands

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

4.3.2 `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.

The closing operation is *not* recursive, i.e. closing a group does not close its children.

_f_isOpen() Is this node open?

_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) Rename this node in place.

Changes the name of a node to *newname* (a string).

_f_move(newparent=None, newname=None, overwrite=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 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.

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_copy(newparent=None, newname=None, overwrite=False, recursive=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 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.

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?

Attribute handling

_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.

_f_delAttr(name) Delete a PyTables attribute from this node.

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

4.4 The Group class

Instances of this class are a grouping structure containing instances of zero or more groups or leaves, together with supporting metadata.

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 name* schema (like in file.root.group1.group2) as discussed in the section 1.2.

A collateral effect of the *natural naming* schema is that names of Group members must be carefully chosen to avoid colliding with existing children node names. For this reason and not to pollute the children namespace, it is explicitly forbidden to assign *normal* attributes to Group instances, and all existing members start with some reserved prefixes, like _f_ (for methods) or _v_ (for instance variables). Any attempt to set a new child node whose name starts with one of these prefixes will raise a ValueError exception.

Another effect of natural naming is that nodes having reserved Python names and other non-allowed Python names (like for example $a or 44) can not be accessed using the node.child syntax. You will be forced to use getattr(node, child) and delattr(node, child) to access them.

You can also make use of the trMap (translation map dictionary) parameter in the openFile function (see section 4.1.2) in order to translate HDF5 names not suited for natural naming into more convenient ones.

4.4.1 Group instance variables

These instance variables are provided in addition to those in Node (see 4.3).

_v_nchildren The number of children hanging from this group.

_v_children Dictionary with all nodes hanging from this group.

_v_groups Dictionary with all groups hanging from this group.

_v_leaves Dictionary with all leaves hanging from this group.

_v_filters Default filter properties for child nodes —see 4.17.1. A shorthand for FILTERS attribute.

4.4.2 Group methods

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
```

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, because these are most used in documentation and examples, and are a bit more powerful than those exposed here.

These methods are provided in addition to those in `Node` (see 4.3).

_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.

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

Copy this node and return the new one.

This method has the behavior described in `Node._f_copy()` (see page 64). 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 4.17.1). 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_iterNodes(classname=None)

Returns an *iterator* yielding all the object nodes hanging from this instance. The nodes are alpha-numerically sorted by its node name. If a *classname* parameter is supplied, it will only return instances of this class (or subclasses of it).

_f_listNodes(classname=None)

Returns a *list* with all the object nodes hanging from this instance. The list is alpha-numerically sorted by node name. If a *classname* parameter is supplied, it will only return instances of this class (or subclasses of it).

_f_walkGroups()

Iterate over the list of Groups (not Leaves) hanging from (and including) *self*. This Group is listed first (pre-order), then each of its child Groups (following an alpha-numerical order) is also traversed, following the same procedure.

_f_walkNodes(classname=None, recursive=True)

Iterate over the nodes in the Group instance. It takes two parameters:

classname *(String)* If supplied, only instances of this class are returned.

recursive *(Integer)* If false, only children hanging immediately after the group are returned. If true, a recursion over all the groups hanging from it is performed.

Example of use:

```
# Recursively print all the arrays hanging from '/'
print "Arrays the object tree '/':"
for array in h5file.root._f_walkNodes("Array", recursive=1):
    print array
```

_f_close()

Close this node in the tree.

This method has the behavior described in Node._f_close() (see page 64). It should be noted that this operation disables access to nodes descending from this group. Therefore, if you want to explicitly close them, you will need to walk the nodes hanging from this group *before* closing it.

_f_copyChildren(dstgroup, overwrite=False, recursive=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 4.4) object or its pathname in string form.

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.

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.

__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 4.2). It does *neither* store a PyTables node attribute; for that, use File.setNodeAttr() (see page 58), Node._f_setAttr() (see page 65) or Node._v_attrs (see page 64).

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 page 58), `Group._f_getChild()` (see page 66) and children dictionaries in the group (if visible).

__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.

__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 page 59) or `Node._f_remove()` (see page 64). It does *neither* delete a PyTables node attribute; for that, use `File.delNodeAttr()` (see page 58), `Node._f_delAttr()` (see page 65) or `Node._v_attrs` (see page 64).

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

__contains__(name)

Is there a child with that *name*?

Returns `True` if the group has a child node (visible or hidden) with the given *name* (a string), `False` otherwise.

__iter__()

Iterate over the children on the group instance. However, this does not accept parameters. 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
```

__str__()

Prints a short description of the `Group` object.

Example of use:

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

__repr__()

Prints a detailed description of the `Group` object.

Example of use:

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

4.5 The Leaf class

The goal of this class is to provide a place to put common functionality of all its descendants as well as provide a way to help classifying objects on the tree. A Leaf object is an end-node, that is, a node that can hang directly from a group object, but that is not a group itself and, thus, it can not have descendants. Right now, the set of end-nodes is composed by Table, Array, CArray, EArray, VLArray and UnImplemented class instances. In fact, all the previous classes inherit from the Leaf class.

4.5.1 Leaf instance variables

These instance variables are provided in addition to those in Node (see 4.3).

shape The shape of data in the leaf.

byteorder The byte ordering of data in the leaf.

filters Filter properties for this leaf —see 4.17.1.

name The name of this node in its parent group (a string). An alias for Node._v_name.

hdf5name The name of this node in the hosting HDF5 file (a string). An alias for Node._v_hdf5name.

objectID The identifier of this node in the hosting HDF5 file. An alias for Node._v_objectID.

attrs The associated AttributeSet instance (see 4.15). An alias for Node._v_attrs.

title A description for this node. An alias for Node._v_title.

4.5.2 Leaf methods

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 objects (i.e. tables) in the same PyTables session, please, call flush() extensively so as to help PyTables to keep memory requirements low.

_f_close(flush=True)

Close this node in the tree.

This method has the behavior described in Node._f_close() (see page 64). Besides that, the optional argument flush tells whether to flush pending data to disk or not before closing.

close(flush=True)

Close this node in the tree.

This method is completely equivalent to _f_close().

isOpen()

Is this node open?

This method is completely equivalent to _f_isOpen().

remove()

Remove this node from the hierarchy.

This method has the behavior described in `Node._f_remove()` (see page 64). Please, note that there is no `recursive` flag since leaves do not have child nodes.

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

Copy this node and return the new one.

This method has the behavior described in `Node._f_copy()` (see page 64). 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 4.17.1). 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 in child leaves 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.

rename(newname)

Rename this node in place.

This method has the behavior described in `Node._f_rename()` (see page 64).

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

Move or rename this node.

This method has the behavior described in `Node._f_move()` (see page 64).

_f_isVisible()

Is this node visible?

This method has the behavior described in `Node._f_isVisible()` (see page 65).

getAttr(name)

Get a PyTables attribute from this node.

This method has the behavior described in `Node._f_getAttr()` (see page 65).

setAttr(name, value)

Set a PyTables attribute for this node.

This method has the behavior described in `Node._f_setAttr()` (see page 65).

delAttr(name)

Delete a PyTables attribute from this node.

This method has the behavior described in `Node._f_delAttr()` (see page 65).

4.6 The Table class

Instances of this class represents table objects in the object tree. It provides methods to read/write data and from/to table objects in the file.

Data can be read from or written to tables by accessing to an special object that hangs from Table. This object is an instance of the Row class (see 4.6.4). See the tutorial sections chapter 3 on how to use the Row interface. The columns of the tables can also be easily accessed (and more specifically, they can be read but not written) by making use of the Column class, through the use of an *extension* of the natural naming schema applied inside the tables. See the section 4.9 for some examples of use of this capability.

Note that this object inherits all the public attributes and methods that Leaf already has.

Finally, during the description of the different methods, there will appear references to a particular object called NestedRecArray. This inherits from numarray.records.RecArray and is designed to keep columns that have nested datatypes. Please, see appendix B for info on these objects.

4.6.1 Table instance variables

description A Description (see 4.8) instance describing the structure of this table.

row The associated Row instance (see 4.6.4).

nrows The number of rows in this table.

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

cols A Cols (see section 4.7) instance that serves as an accessor to Column (see section 4.9) objects.

colnames A tuple containing the (possibly nested) names of the columns in the table.

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

colstypes Maps the name of a column to its data string type.

colshapes Maps the name of a column to it shape.

colitemsizes Maps the name of a column to the size of its base items.

coldflts Maps the name of a column to its default.

colindexed Is the column which name is used as a key indexed? (dictionary)

indexed Does this table have any indexed columns?

indexprops Index properties for this table (an IndexProps instance, see 4.17.2).

flavor The default flavor for this table. This determines the type of objects returned during input (i.e. read) operations. It can take the *"numarray"* (default) or *"numpy"* values. Its value is derived from the _v_flavor attribute of the IsDescription metaclass (see 4.16.1) or, if the table has been created directly from a numarray or NumPy object, the flavor is set to the appropriate value.

4.6.2 Table methods

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 4.17.4) is returned. If it is not of an enumerated type, a TypeError is raised. If the column does not exist, a KeyError is raised.

append(rows)

Append a series of rows to this `Table` instance. *rows* is an object that can keep the rows to be append in several formats, like a `NestedRecArray` (see appendix B), a `RecArray`, a `NumPy` object, a list of tuples, list of `Numeric`/`numarray`/`NumPy` objects, string, Python buffer or `None` (no append will result). Of course, this *rows* object has to be compliant with the underlying format of the `Table` instance or a `ValueError` will be issued.

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()
```

col(name)

Get a column from the table.

If a column called `name` exists in the table, it is read and returned as a `numarray` object, or as a `NumPy` object (whatever is more appropriate 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 read()` (see 4.6.2) method.

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding `Row` (see section 4.6.4) instances built from rows in table. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the `__iter__()` special method in section 4.6.3 for a shorter way to call this iterator.

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`. If you do not specify neither *start* nor *stop*, then **all the rows** in the object are selected.

Example of use:

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

Note: This iterator can be nested (see example in section 4.6.2).

itersequence(sequence, sort=True)

Iterate over a *sequence* of row coordinates.

sequence Can be any object that supports the __getitem__ special method, like lists, tuples, Numeric/NumPy/numarray objects, etc.

sort If true, means that *sequence* will be sorted out so that the I/O process would get better performance. If your sequence is already sorted or you don't want to sort it, put this parameter to 0. The default is to sort the *sequence*.

Note: This iterator can be nested (see example in section 4.6.2).

read(start=None, stop=None, step=1, field=None, flavor=None)

Returns the actual data in Table. If *field* is not supplied, it returns the data as a NestedRecArray (see appendix B) object table.

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*. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

The rest of the parameters are described next:

field If specified, only the column *field* is returned as an homogeneous numarray/NumPy/Numeric object, depending on the *flavor*. If this is not supplied, all the fields are selected and a NestedRecArray (see appendix B) or NumPy object is returned. Nested fields can be specified in the *field* parameter by using a '/' character as a separator between fields (e.g. Info/value).

flavor Passing a *flavor* parameter make an additional conversion to happen in the default returned object. *flavor* can have any of the next values: "numarray" "numpy", "python" or "numeric" (only if *field* has been specified). If *flavor* is not specified, then it will take the value of self.flavor.

readCoordinates(coords, field=None, flavor=None)

Read a set of rows given their indexes into an in-memory object.

This method works much like the read() method (see 4.6.2), but it uses a sequence (coords) of row indexes to select the wanted columns, instead of a column range.

It returns the selected rows in a NestedRecArray object (see appendix B). If flavor is provided, an additional conversion to an object of this flavor is made, just as in read().

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

Modify a series of rows in the [start:stop:step] *extended slice* range. If you pass None to *stop*, all the rows existing in *rows* will be used.

rows can be either a *recarray* or a structure that is able to be converted to any of them and compliant with the table format.

Returns the number of modified rows.

It raises an ValueError in case the rows parameter could not be converted to an object compliant with table description.

It raises an IndexError in case the modification will exceed the length of the table.

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

Modify a series of rows in the [start:stop:step] *extended slice* row range. If you pass None to *stop*, all the rows existing in *column* will be used.

column can be either a NestedRecArray (see appendix B), RecArray, numarray, NumPy object, list or tuple that is able to be converted into a NestedRecArray compliant with the specified *colname* column of the table.

colname specifies the column name of the table to be modified.

Returns the number of modified rows.

It raises an `ValueError` in case the *column* parameter could not be converted into an object compliant with `column` description.

It raises an `IndexError` in case the modification will exceed the length of the table.

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

Modify a series of rows in the `[start:stop:step]` *extended slice* row range. If you pass `None` to *stop*, all the rows existing in *columns* will be used.

columns can be either a `NestedRecArray` (see appendix B), `RecArray`, a `NumPy` object, a list of arrays or list or tuples (the columns) that are able to be converted to a `NestedRecArray` compliant with the specified column *names* subset of the table format.

names specifies the column names of the table to be modified.

Returns the number of modified rows.

It raises an `ValueError` in case the *columns* parameter could not be converted to an object compliant with table description.

It raises an `IndexError` in case the modification will exceed the length of the table.

removeRows(start, stop=None)

Removes a range of rows in the table. If only *start* is supplied, 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 implement it 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). It accepts, likewise *start*, negative values. A special value of `None` (the default) means removing just the row supplied in start.

removeIndex(index)

Remove the index associated with the specified column. Only `Index` instances (see 4.17.3) are accepted as parameter. This index can be recreated again by calling the `createIndex` (see 4.9.2) method of the appropriate `Column` object.

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 section 4.17.2) and want to update the indexes on it.

reIndex()

Recompute all the existing indexes in 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.

reIndexDirty()

Recompute the existing indexes in table, but *only* if they are dirty. This can be useful when you have set the `reindex` parameter to 0 in `IndexProps` constructor (see 4.17.2) for the table and want to update the indexes after a invalidating index operation (`Table.removeRows`, for example).

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

Iterate over values fulfilling a condition.

This method returns an iterator yielding Row (see 4.6.4) instances built from rows in the table that satisfy the given condition over a column. If that column is indexed, its index will be used in order to accelerate the search. Else, the *in-kernel* iterator (with has still better performance than standard Python selections) will be chosen instead. Please, check the section 5.2 for more information about the performance of the different searching modes.

Moreover, if a range is supplied (i.e. some of the start, stop or step parameters are passed), only the rows in that range *and* fulfilling the condition are returned. 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.

You can mix this method with standard Python selections in order to have 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=[]
for row in table.where(0 < table.cols.col1 < 0.3, step=5):
    if row['col2'] <= 20:
        passvalues.append(row['col3'])
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(rout.cols.pressure < 16):
    for q in rout.where(rout.cols.pressure < 9):
        for n in rout.where(rout.cols.energy < 10):
            print "pressure, energy:", p['pressure'],n['energy']
```

In this example, the iterators returned by where() has been nested, but in fact, you can use any of the other reading iterators that the Table object offers. Look at examples/nested-iter.py for the full code.

whereAppend(dstTable, condition, 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 where() method (see 4.6.2).

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

getWhereList(condition, flavor=None)

Get the row coordinates that fulfill the *condition* parameter. This method will take advantage of an indexed column to speed-up the search.

flavor is the desired type of the returned list. It can take the "numarray", "numpy", "numeric" or "python" values. The default is returning an object of the same flavor than self.flavor.

4.6.3 Table special methods

Following are described the methods that automatically trigger actions when a Table instance is accessed in a special way (e.g., table["var2"] will be equivalent to a call to table.__getitem__("var2")).

__iter__()

It returns the same iterator than `Table.iterrows(0,0,1)`. However, this does not accept parameters.

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 example in section 4.6.2).

__getitem__(key)

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

If the `key` argument is an integer, the corresponding table row is returned as a `tables.nestedrecords.NestedRecord` object. If `key` is a slice, the range of rows determined by it is returned as a `tables.nestedrecords.NestedRecArray` object.

Using a string as `key` to get a column is supported but deprecated. Please use the `col()` (see 4.6.2) method.

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 4.6.2) method.

__setitem__(key, value)

It takes different actions depending on the type of the `key` parameter:

`key` is an `Integer` The corresponding table row is set to *value*. *value* must be a `List` or `Tuple` capable of being converted to the table field format.

`key` is a `Slice` The row slice determined by key is set to *value*. *value* must be a `NestedRecArray` object or a `RecArray` object or a list of rows capable of being converted to the table field format.

Example of use:

```
# Modify just one existing row
table[2] = [456,'db2',1.2]
# Modify two existing rows
rows = numarray.records.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 = numarray.records.array([[457,'db1',1.2],[6,'de2',1.3]],
formats="i4,a3,f8")
table.modifyRows(start=1, step=2, rows=rows)
```

4.6.4 The Row class

This class is used to fetch and set values on the table fields. It works very much like a dictionary, where the keys are the field names of the associated table and the values are the values of those fields in a specific row.

This object turns out to actually be an extension type, so you won't be able to access its documentation interactively. However, you will be able to access some of its internal attributes through the use of Python properties. In addition, there are some important methods that are useful for adding and modifying values in tables.

Row attributes

nrow Property that returns the current row number in the table. It is useful to know which row is being dealt with in the middle of a loop or iterator.

Row methods

append() Once you have filled the proper fields for the current row, calling this method actually append these new data to the disk (actually data are written to the output buffer).

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()
```

Please, note that, after 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 can be in internal buffers.

update() This allows you to modify values of your tables when you are in the middle of table iterators, like `Table.iterrows()` (see 4.6.2) or `Table.where()` (see 4.6.2). Once you have filled the proper fields for the current row, calling this method actually commits these data to the disk (actually data are written to the output buffer).

Example of use:

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

which modifies every tenth row in table. Or:

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

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

4.7 The `Cols` class

This class is used as an *accessor* to the table columns following the natural name convention, so that you can access the different columns because there exists one attribute with the name of the columns for each associated column, which can be a `Column` instance (non-nested column) or another `Cols` instance (nested column).

Columns under a `Cols` accessor can be accessed as attributes of it. 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 and so forth.

4.7.1 `Cols` instance variables

_v_colnames A list of the names of the columns (or nested columns) hanging directly from this `Cols` instance. The order of the names matches the order of their respective columns in the containing table.

_v_colpathnames A list of the complete pathnames of the columns hanging directly from this `Cols` instance. If the table does not contain nested columns, this is exactly the same as `_v_colnames` attribute.

_v_table The parent `Table` instance.

_v_desc The associated Description 4.9 instance.

4.7.2 `Cols` methods

_f_col(colname)

Return a handler to the *colname* column. If *colname* is a nested column, a `Cols` instance is returned. If *colname* is a non-nested column a `Column` object is returned instead.

__getitem__(key)

Get a row or a range of rows from the `Cols` accessor.

If the `key` argument is an integer, the corresponding `Cols` row is returned as a `tables.nestedrecords.NestedRecord` object. If `key` is a slice, the range of rows determined by it is returned as a `tables.nestedrecords.NestedRecArray` object.

Using a string as `key` to get a column is supported but deprecated. Please use the `col()` (see 4.6.2) method.

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 read() (see 4.6.2) method.

__setitem__(key)

Set a row or a range of rows to the Cols accessor.

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

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 modifyRows() and modifyColumn() (see 4.6.2 and 4.6.2) methods.

4.8 The Description class

The instances of the Description class provide a description of the structure of a table.

An instance of this class is automatically bound to Table (see 4.6) objects when they are created. It provides a browseable representation of the structure of the table, made of non-nested (Col —see 4.16.2) and nested (Description) columns. It also contains information that will allow you to build NestedRecArray (see appendix B) objects suited for the different columns in a table (be they nested or not).

Column descriptions (see Col class in 4.16.2) under a description can be accessed as attributes of it. 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.

4.8.1 Description instance variables

_v_name The name of this description instance. If description is the root of the nested type (or the description of a flat table), its name will be the empty string ('').

_v_names A list of the names of the columns hanging directly from this description instance. The order of the names matches the order of their respective columns in the containing description.

_v_pathnames A list of the pathnames of the columns hanging directly from this description. If the table does not contain nested columns, this is exactly the same as _v_names attribute.

_v_nestedNames A nested list of the names of all the columns hanging directly from this description instance. You can use this for the names argument of NestedRecArray factory functions.

_v_nestedFormats A nested list of the numarray string formats (and shapes) of all the columns hanging directly from this description instance. You can use this for the formats argument of NestedRecArray factory functions.

_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 for the `descr` argument of `NestedRecArray` factory functions.

_v_types A dictionary mapping the names of non-nested columns hanging directly from this description instance to their respective numarray types.

_v_stypes A dictionary mapping the names of non-nested columns hanging directly from this description instance to their respective string types.

_v_shapes A dictionary mapping the names of non-nested columns hanging directly from this description instance to their respective shapes.

_v_dflts A dictionary mapping the names of non-nested columns hanging directly from this description instance to their respective default values. Please, note that all the default values are kept internally as numarray objects.

_v_colObjects A dictionary mapping the names of the columns hanging directly from this description instance to their respective descriptions (`Col` —see 4.16.2— or `Description` —see 4.8— instances).

_v_itemsizes A dictionary mapping the names of non-nested columns hanging directly from this description instance to their respective item size (in bytes).

_v_nestedlvl The level of the description in the nested datatype.

4.8.2 `Description` methods

_v_walk(type='All') Iterate over nested columns.

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

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

4.9 The `Column` class

Each instance of this class is associated with one column of every table. These instances are mainly used to fetch and set actual data from the table columns, but there are a few other associated methods to deal with indexes.

4.9.1 `Column` instance variables

table The parent `Table` instance.

name The name of the associated column.

pathname The complete pathname of the associated column. This is mainly useful in nested columns; for non-nested ones this value is the same a `name`.

type The data type of the column.

shape The shape of the column.

index The associated `Index` object (see 4.17.3) to this column (`None` if does not exist).

dirty Whether the index is dirty or not (property).

4.9.2 `Column` methods

`createIndex()`

Create an `Index` (see 4.17.3) object for this column.

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 want to rebuild it.

reIndexDirty()

Recompute the existing index only if it is dirty. This can be useful when you have set the reindex parameter to 0 in IndexProps constructor (see 4.17.2) for the table and want to update the column's index after a invalidating index operation (Table.removeRows, for example).

removeIndex()

Delete the associated column's index. After doing that, you will loose the indexation information on disk. However, you can always re-create it using the createIndex() method (see 4.9.2).

4.9.3 Column special methods

__getitem__(key)

Returns a column element or slice. It takes different actions depending on the type of the *key* parameter:

key is an Integer The corresponding element in the column is returned as a scalar object or as a numarray object, depending on its shape.

key is a Slice The row range determined by this slice is returned as a numarray object.

Example of use:

```
print "Column handlers:"
for name in table.colnames:
    print table.cols[name]
print
print "Some selections:"
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.lati[1:3]-->", table.cols.lati[1:3]
print "Select table.cols.pressure[:]-->", table.cols.pressure[:]
print "Select table.cols['temperature'][:]-->", table.cols['temperature'][:]
```

and the output of this for a certain arbitrary table is:

```
Column handlers:
/table.cols.name (Column(1,), CharType)
/table.cols.lati (Column(2,), Int32)
/table.cols.longi (Column(1,), Int32)
/table.cols.pressure (Column(1,), Float32)
/table.cols.temperature (Column(1,), Float64)

Some selections:
Select table.cols.name[1]--> Particle:     11
Select table.cols.name[1:2]--> ['Particle:     11']
Select table.cols.lati[1:3]--> [[11 12]
 [12 13]]
Select table.cols.pressure[:]--> [  90.  110.  132.]
Select table.cols['temperature'][:]--> [ 100.  121.  144.]
```

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

__setitem__(key, value)

It takes different actions depending on the type of the `key` parameter:

key is an `Integer` The corresponding element in the column is set to *value*. *value* must be a scalar or `numarray`/`NumPy` object, depending on column's shape.

key is a `Slice` The row slice determined by *key* is set to *value*. *value* must be a list of elements or a `numarray`/`NumPy`.

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 = numarray.records.fromarrays([[2,3]], formats="i4")
table.modifyColumns(start=1, step=2, columns=columns, names=["col1"])
```

4.10 The `Array` class

Represents an array on file. It provides methods to write/read data to/from array objects in the file. This class does not allow you to enlarge the datasets on disk; see the `EArray` descendant in section 4.12 if you want enlargeable dataset support and/or compression features. See also `CArray` in section 4.11

The array data types supported are the same as the set provided by the `numarray` package. For details of these data types see appendix A, or the `numarray` reference manual (Greenfield *et al.*).

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, or if you saved a `NumPy` array, you will get a `NumPy` object.

Note that this object inherits all the public attributes and methods that `Leaf` already provides.

4.10.1 `Array` instance variables

flavor The object representation for this array. It can be any of *"numarray"*, *"numpy"*, *"numeric"* or *"python"* values.

nrows The length of the first dimension of the array.

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

type The type class of the represented array.

stype The string type of the represented array.

itemsize The size of the base items. Specially useful for `CharType` objects.

4.10.2 `Array` methods

Note that, as this object has no internal I/O buffers, it is not necessary to use the flush() method inherited from `Leaf` in order to save its internal state to disk. When a writing method call returns, all the data is already on disk.

getEnum()

Get the enumerated type associated with this array.

If this array is of an enumerated type, the corresponding Enum instance (see 4.17.4) is returned. If it is not of an enumerated type, a TypeError is raised.

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding numarray instances built from rows in array. The return rows are taken from the first dimension in case of an Array and CArray instance and the enlargeable dimension in case of an EArray instance. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the and __iter__() special methods in section 4.10.3 for a shorter way to call this iterator.

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. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

Example of use:

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

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

Read the array from disk and return it as a numarray (default) object, or an object with the same original *flavor* that it was saved. It accepts *start*, *stop* and *step* parameters to select rows (the first dimension in the case of an Array and CArray instance and the *enlargeable* dimension in case of an EArray) for reading.

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. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

4.10.3 Array special methods

Following are described the methods that 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))).

__iter__()

It returns the same iterator than Array.iterrows(0,0,1). However, this does not accept parameters.

Example of use:

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

Which is equivalent to:

```
result = [ row[2] for row in array.iterrows(0, 0, 1) ]
```

__getitem__(key)

It returns a numarray (default) object (or an object with the same original *flavor* that it was saved) containing the slice of rows stated in the key parameter. The set of allowed tokens in key is the same as extended slicing in python (the Ellipsis token included).

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
```

__setitem__(key, value)

Sets an Array element, row or extended slice. It takes different actions depending on the type of the `key` parameter:

key is an integer: The corresponding row is assigned to value. If needed, this `value` is broadcasted to fit the specified row.

key is a slice: The row slice determined by it is assigned to `value`. If needed, this `value` is broadcasted to fit in the desired range. If the slice to be updated exceeds the actual shape of the array, only the values in the existing range are updated, i.e. the index error will be silently ignored. If `value` is a multidimensional object, then its shape must be compatible with the slice specified in `key`, otherwise, a `ValueError` will be issued.

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))
```

4.11 The `CArray` class

This is a child of the `Array` class (see 4.10) and as such, `CArray` represents an array on the file. The difference is that `CArray` has a chunked layout and, as a consequence, it also supports compression. You can use this class to easily save or load array (or array slices) objects to or from disk, with compression support included.

4.11.1 `CArray` instance variables

In addition to the attributes that `CArray` inherits from `Array`, it supports some more that provide information about the filters used.

atom An `Atom` (see 4.16.3) instance representing the shape, type and flavor of the atomic objects to be saved.

4.11.2 Example of use

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

```
import numarray
import tables

fileName = 'carray1.h5'
shape = (200,300)
atom = tables.UInt8Atom(shape = (128,128))
filters = tables.Filters(complevel=5, complib='zlib')
```

```
h5f = tables.openFile(fileName,'w')
ca = h5f.createCArray(h5f.root, 'carray', shape, atom, filters=filters)
# Fill a hyperslab in ca. The array will be converted to UInt8 elements
ca[10:60,20:70] = numarray.ones((50,50))
h5f.close()

# Re-open a 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 Jun 16 10:47:18 2005'
Object Tree:
/ (RootGroup) ''
/carray (CArray(200L, 300L)) ''

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

4.12 The `EArray` class

This is a child of the `Array` class (see 4.10) and as such, `EArray` represents an array on the file. The difference is that `EArray` allows to enlarge datasets along any single dimension[1] you select. Another important difference is that it also supports compression.

So, in addition to the attributes and methods that `EArray` inherits from `Array`, it supports a few more that provide a way to enlarge the arrays on disk. Following are described the new variables and methods as well as some that already exist in `Array` but that differ somewhat on the meaning and/or functionality in the `EArray` context.

4.12.1 `EArray` instance variables

atom An `Atom` (see 4.16.3) instance representing the shape, type and flavor of the atomic objects to be saved. One of the dimensions of the shape is 0, meaning that the array can be extended along it.

extdim The enlargeable dimension, i.e. the dimension this array can be extended along.

nrows The length of the enlargeable dimension of the array.

4.12.2 `EArray` methods

getEnum()

Get the enumerated type associated with this array.

If this array is of an enumerated type, the corresponding `Enum` instance (see 4.17.4) is returned. If it is not of an enumerated type, a `TypeError` is raised.

[1] In the future, multiple enlargeable dimensions might be implemented as well.

append(sequence)

Appends a `sequence` to the underlying dataset. Obviously, this sequence must have the same type as the `EArray` instance; otherwise a `TypeError` is issued. In the same way, the dimensions of the `sequence` have to conform to those of `EArray`, that is, all the dimensions have to be the same except, of course, that of the enlargeable dimension which can be of any length (even 0!).

Example of use (code available in `examples/earray1.py`):

```
import tables
from numarray import strings

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

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

and the output is:

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

4.13 The `VLArray` class

Instances of this class represents array objects in the object tree with the property that their rows can have a **variable** number of (homogeneous) elements (called *atomic* objects, or just *atoms*). Variable length arrays (or *VLA's* for short), similarly to `Table` instances, can have only one dimension, and likewise `Table`, the compound elements (the *atoms*) of the rows of `VLArrays` can be fully multidimensional objects.

`VLArray` provides methods to read/write data from/to variable length array objects residents on disk. Also, note that this object inherits all the public attributes and methods that `Leaf` already has.

4.13.1 `VLArray` instance variables

atom An `Atom` (see 4.16.3) instance representing the shape, type and flavor of the atomic objects to be saved.

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

nrows The total number of rows.

4.13.2 `VLArray` methods

getEnum()

Get the enumerated type associated with this array.

If this array is of an enumerated type, the corresponding Enum instance (see 4.17.4) is returned. If it is not of an enumerated type, a TypeError is raised.

append(sequence, *objects)

Append objects in the sequence to the array.

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

Example of use (code available in examples/vlarray1.py):

```
import tables
from numpy import *   # or, from numarray import *

# Create a VLArray:
fileh = tables.openFile("vlarray1.h5", mode = "w")
vlarray = fileh.createVLArray(fileh.root, 'vlarray1',
tables.Int32Atom(flavor="numpy"),
                 "ragged array of ints", Filters(complevel=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:
for x in vlarray:
    print vlarray.name+"["+str(vlarray.nrow)+"]-->", x

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

The output of the previous program looks like this:

```
vlarray1[0]--> [5 6]
vlarray1[1]--> [5 6 7]
vlarray1[2]--> [5 6 9 8]
```

The objects argument is only retained for backwards compatibility; please do *not* use it.

iterrows(start=None, stop=None, step=1)

Returns an iterator yielding one row per iteration. If a range is supplied (i.e. some of the *start*, *stop* or *step* parameters are passed), only the appropriate rows are returned. Else, all the rows are returned. See also the __iter__() special methods in section 4.13.3 for a shorter way to call this iterator.

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. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

Example of use:

```
for row in vlarray.iterrows(step=4):
print vlarray.name+"["+str(vlarray.nrow)+"]-->", row
```

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

Returns the actual data in `VLArray`. As the lengths of the different rows are variable, the returned value is a python list, with as many entries as specified rows in the range parameters.

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`. If you do not specify neither *start* nor *stop*, then all the rows in the object are selected.

4.13.3 `VLArray` special methods

Following are described the methods that 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))`.

__iter__()

It returns the same iterator than `VLArray.iterrows(0,0,1)`. However, this does not accept parameters.

Example of use:

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

Which is equivalent to:

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

__getitem__(key)

It returns the slice of rows determined by `key`, which can be an integer index or an extended slice. The returned value is a list of objects of type `array.atom.type`.

Example of use:

```
list1 = vlarray[4]
list2 = vlarray[4:1000:2]
```

__setitem__(keys, value)

Updates a vlarray row described by `keys` by setting it to `value`. Depending on the value of `keys`, the action taken is different:

`keys` is an integer: It refers to the number of row to be modified. The `value` object must be type and shape compatible with the object that exists in the vlarray row.

`keys` is a tuple: The first element refers to the row to be modified, and the second element to the range (so, it can be an integer or an slice) of the row that will be updated. As above, the `value` object must be type and shape compatible with the object specified in the vlarray row *and* range.

Note: When updating `VLStrings` (codification UTF-8) or `Objects` atoms, there is a problem: one can only update values with *exactly* the same bytes than in the original row. With UTF-8 encoding this is problematic because, for instance, '`c`' takes 1 byte, but '`ç`' takes two. The same applies when using `Objects` atoms, because when cPickle applies to a class instance (for example), it does not guarantee to return the same number of bytes than over other instance, even of the same class than the former. These facts effectively limit the number of objects than can be updated in `VLArray`s.

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
```

4.14 The UnImplemented class

Instances of this class represents an unimplemented dataset in a generic HDF5 file. When reading such a 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* and/or *dataspaces* in some dataset might not be supported by PyTables yet. In such a case, this dataset will be mapped into the UnImplemented class and hence, the user will still be able to build the complete object tree of this generic HDF5 file, as well as enabling the access (both read and *write*) of the attributes of this dataset and some metadata. Of course, the user won't 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 would not be supported by PyTables. However, if you are really interested in having access to an unimplemented dataset, please, get in contact with the developer team.

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

4.15 The AttributeSet class

Represents the set of attributes of a node (Leaf or Group). It provides methods to create new attributes, open, rename or delete existing ones.

Like in Group instances, AttributeSet instances make use of the *natural naming* convention, i.e. you can access the attributes on disk like if they were *normal* AttributeSet attributes. This offers the user a very convenient way to access (but also to set and delete) node attributes by simply specifying them like a *normal* attribute class.

Caveat emptor: All Python data types are supported. In particular, multidimensional numarray objects are saved natively as multidimensional objects in the HDF5 file. Python strings are also saved natively as HDF5 strings, and loaded back as Python strings. However, the rest of the data types including the Python scalar ones (i.e. Int, Long and Float) and more general objects (like NumPy or Numeric) are serialized using cPickle, so you will be able to correctly retrieve them only from a Python-aware HDF5 library. So, if you want to save Python scalar values and be able to read them with generic HDF5 tools, you should make use of scalar numarray objects (for example numarray.array(1, type=numarray.Int64)). In the same way, attributes in HDF5 native files will be always mapped into numarray objects. Specifically, a multidimensional attribute will be mapped into a multidimensional numarray and an scalar will be mapped into a scalar numarray (for example, an attribute of type H5T_NATIVE_LLONG will be read and returned as a numarray.array(X, type=numarray.Int64) scalar).

One more warning: 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 tables
>>>
>>> class MyClass(object):
...   foo = 'bar'
...
>>> # An object of my custom class.
... myObject = MyClass()
```

```
>>>
>>> h5f = tables.openFile('test.h5', 'w')
>>> h5f.root._v_attrs.obj = myObject  # store the object
>>> print h5f.root._v_attrs.obj.foo  # retrieve it
bar
>>> h5f.close()
>>>
>>> # Delete class of stored object and reopen the file.
... del MyClass, myObject
>>>
>>> h5f = tables.openFile('test.h5', 'r')
>>> print h5f.root._v_attrs.obj.foo
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'str' object has no attribute 'foo'
>>> # Let us inspect the object to see what is happening.
... print repr(h5f.root._v_attrs.obj)
'ccopy_reg\n_reconstructor\np1\n(c__main__\nMyClass\np2\nc__builtin__\nobject\np3\nNtRp4\n
>>> # Maybe unpickling the string will yield more information:
... import cPickle
>>> cPickle.loads(h5f.root._v_attrs.obj)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
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()
```

4.15.1 `AttributeSet` instance variables

_v_node The parent node instance.

_v_attrnames List with all attribute names.

_v_attrnamessys List with system attribute names.

_v_attrnamesuser List with user attribute names.

4.15.2 `AttributeSet` methods

Note that this class defines the `__setattr__`, `__getattr__` and `__delattr__` and they work as normally intended. Any scalar (string, ints or floats) attribute is supported natively as an attribute. However, `(c)Pickle` is automatically used so as to serialize other kind of objects (like lists, tuples, dicts, small NumPy/Numeric/numarray objects, ...) that you might want to save. If an attribute is set on a target node that already has a large number of attributes, a `PerformanceWarning` will be issued.

With these special methods, you can access, 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
```

_f_copy(where) Copy the user attributes (as well as *certain* system attributes) to *where* object. *where* has to be a `Group` or `Leaf` instance.

_f_list(attrset = "user") Return a list of attribute names of the parent node. *attrset* selects the attribute set to be used. A `user` value returns only the user attributes and this is the default. `sys` returns only the system attributes. `all` returns both the system and user attributes.

_f_rename(oldattrname, newattrname) Rename an attribute.

4.16 Declarative classes

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

4.16.1 The `IsDescription` class

This class is designed to be used as an easy, yet meaningful way to describe the properties of `Table` objects through the definition of *derived classes* that inherit properties from it. 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` (see section 4.16.2) class. Nested columns can be described by using classes derived from `IsDescription` or instances of it. Derived classes can be declared in place (in which case the column takes the name of the class) or referenced by name, and they can have a `_v_pos` special attribute which sets the position of the nested column among its sibling columns.

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.

See below for a complete list of the special attributes that can be specified to complement the *metadata* of an `IsDescription` class.

`IsDescription` special attributes

_v_flavor The flavor of the table. It can take *"numarray"* (default) or *"numpy"* values. This determines the type of objects returned during input (i.e. read) operations.

_v_indexprops An instance of the `IndexProps` class (see section 4.17.2). You can use this to alter the properties of the index creation process for a table.

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

4.16.2 The `Col` class and its descendants

The `Col` class is used as a mean to declare the different properties of a table column. In addition, a series of descendant classes are offered in order to make these column descriptions easier to the user. In general, it is recommended to use these descendant classes, as they are more meaningful when found in the middle of the code.

`Col` instance attributes

type The type class of the column.

stype The string type of the column.

recarrtype The string type, in `RecArray` format, of the column.

shape The shape of the column.

itemsize The size of the base items. Specially useful for `StringCol` objects.

indexed Whether this column is meant to be indexed or not.

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

_v_name The name of this column

_v_pathname The complete pathname of the column. This is mainly useful in nested columns; for non-nested ones this value is the same a `_v_name`.

`Col` methods

None.

`Col` constructors

A description of the different constructors with their parameters follows:

Col(dtype="Float64", shape=1, dflt=None, pos=None, indexed=0) Declare the properties of a `Table` column.

dtype The data type for the column. All types listed in appendix A are valid data types for columns. The type description is accepted both in string-type format and as a numarray data type.

shape An integer or a tuple, that specifies the number of *dtype* items for each element (or shape, for multidimensional elements) of this column. For `CharType` columns, the last dimension is used as the length of the character strings. However, for this kind of objects, the use of `StringCol` subclass is strongly recommended.

dflt The default value for elements of this column. If the user does not supply a value for an element while filling a table, this default value will be written to disk. If the user supplies an scalar value for a multidimensional column, this value is automatically *broadcasted* to all the elements in the column cell. If *dflt* is not supplied, an appropriate zero value (or *null* string) will be chosen by default. Please, note that all the default values are kept internally as numarray objects.

pos By default, columns are arranged in memory following an alpha-numerical order of the column names. In some situations, however, it is convenient to impose a user defined ordering. *pos* parameter allows the user to force the desired ordering.

indexed Whether this column should be indexed for better performance in table selections.

StringCol(length=None, dflt=None, shape=1, pos=None, indexed=0) Declare a column to be of type `CharType`. The *length* parameter sets the length of the strings. The meaning of the other parameters are like in the `Col` class.

BoolCol(dflt=0, shape=1, pos=None, indexed=0) Define a column to be of type `Bool`. The meaning of the parameters are the same of those in the `Col` class.

IntCol(dflt=0, shape=1, itemsize=4, sign=1, pos=None, indexed=0) Declare a column to be of type `IntXX`, depending on the value of *itemsize* parameter, that sets the number of bytes of the integers in the column. *sign* determines whether the integers are signed or not. The meaning of the other parameters are the same of those in the `Col` class.

This class has several descendants:

Int8Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `Int8`.

UInt8Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `UInt8`.

Int16Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `Int16`.

UInt16Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `UInt16`.

Int32Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `Int32`.

UInt32Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `UInt32`.

Int64Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `Int64`.

UInt64Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `UInt64`.

FloatCol(dflt=0.0, shape=1, itemsize=8, pos=None, indexed=0) Define a column to be of type `FloatXX`, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the floats in the column and the default is 8 bytes (double precision). The meaning of the other parameters are the same as those in the `Col` class.

This class has two descendants:

Float32Col(dflt=0.0, shape=1, pos=None, indexed=0) Define a column of type `Float32`.

Float64Col(dflt=0.0, shape=1, pos=None, indexed=0) Define a column of type `Float64`.

ComplexCol(dflt=0.+0.j, shape=1, itemsize=16, pos=None) Define a column to be of type `ComplexXX`, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the complex types in the column and the default is 16 bytes (double precision complex). The meaning of the other parameters are the same as those in the `Col` class.

This class has two descendants:

Complex32Col(dflt=0.+0.j, shape=1, pos=None) Define a column of type `Complex32`.

Float64Col(dflt=0+0.j, shape=1, pos=None) Define a column of type `Complex64`.

`ComplexCol` columns and its descendants do not support indexation.

TimeCol(dflt=0, shape=1, itemsize=8, pos=None, indexed=0) Define a column to be of type *Time*. Two kinds of time columns are supported depending on the value of `itemsize`: 4-byte signed integer and 8-byte double precision floating point columns (the default ones). The meaning of the other parameters are the same as those in the `Col` class.

Time columns have a special encoding in the HFD5 file. See appendix A for more information on those types.

This class has two descendants:

Time32Col(dflt=0, shape=1, pos=None, indexed=0) Define a column of type `Time32`.

Time64Col(dflt=0.0, shape=1, pos=None, indexed=0) Define a column of type `Time64`.

EnumCol(enum, dflt, dtype='UInt32', shape=1, pos=None, indexed=False) Description of a column of an enumerated type.

Instances of this class describe a table column which stores enumerated values. Those values belong to an enumerated type, defined by the first argument (`enum`) in the constructor of `EnumCol`, which accepts the same kinds of arguments as `Enum` (see 4.17.4). The enumerated type is stored in the `enum` attribute of the column.

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. Once the column is created, the corresponding concrete value is stored in its `dflt` attribute. If the name does not match any value in the enumerated type, a `KeyError` is raised.

A numarray data type might be specified in order to determine the base type used for storing the values of enumerated values in memory and disk. The data type must be able to represent each and every concrete value in the enumeration. If it is not, a `TypeError` is raised. The default base type is unsigned 32-bit integer, which is sufficient for most cases.

The `stype` attribute of enumerated columns is always `'Enum'`, while the `type` attribute is the data type used for storing concrete values.

The shape, position and indexed attributes of the column are treated as with other column description objects (see 4.16.2).

4.16.3 The `Atom` class and its descendants.

The `Atom` class is a descendant of the `Col` class (see 4.16.2) and is meant to declare the different properties of the *base element* (also known as *atom*) of `CArray`, `EArray` and `VLArray` objects. The `Atom` instances have the property that their length is always the same. However, you can grow objects along the extensible dimension in the case of `EArray` or put a variable number of them on a `VLArray` row. Moreover, the atoms 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. In general, it is recommended to use these descendant classes, as they are more meaningful when found in the middle of the code.

`Atom` instance variables

In addition to the variables that it inherits from the `Col` class, it has the next additional attributes:

flavor The object representation for this atom. See below on constructors description for `Atom` class the possible values it can take.

`Atom` methods

atomsize() Returns the total length, in bytes, of the element base atom. If its shape is has one zero element on it (for use in `EArrays`, for example), this is replaced by an one in order to compute the atom size correctly.

`Atom` constructors

A description of the different constructors with their parameters follows:

Atom(dtype="Float64", shape=1, flavor="numarray") Define properties for the base elements of `CArray`, `EArray` and `VLArray` objects.

- **dtype** The data type for the base element. See the appendix A for a relation of data types supported. The type description is accepted both in string-type format and as a numarray data type.
- **shape** In a `EArray` context, it is a **tuple** specifying the shape of the object, and one (and only one) of its dimensions **must** be 0, meaning that the `EArray` object will be enlarged along this axis. In the case of a `VLArray`, it can be an integer with a value of 1 (one) or a tuple, that specifies whether the atom is an scalar (in the case of a 1) or has multiple dimensions (in the case of a tuple). For `CharType` elements, the last dimension is used as the length of the character strings. However, for this kind of objects, the use of `StringAtom` subclass is strongly recommended.
- **flavor** The object representation for this atom. It can be any of *"numarray"*, *"numpy"* or *"python"* for the character types and *"numarray"*, *"numpy"*, *"numeric"* or *"python"* for the numerical types. If specified, the read atoms will be converted to that specific flavor. If not specified, the atoms will remain in their native format (i.e. `numarray`).

StringAtom(shape=1, length=None, flavor="numarray") Define an atom to be of `CharType` type. The meaning of the *shape* parameter is the same as in the `Atom` class. *length* sets the length of the strings atoms. *flavor* can be whether `"numarray"`, `"numpy"` or `"python"`. Unicode strings are not supported by this type; see the `VLStringAtom` class if you want Unicode support (only available for `VLAtom` objects).

BoolAtom(shape=1, flavor="numarray") Define an atom to be of type `Bool`. The meaning of the parameters are the same of those in the `Atom` class.

IntAtom(shape=1, itemsize=4, sign=1, flavor="numarray") Define an atom to be of type `IntXX`, depending on the value of *itemsize* parameter, that sets the number of bytes of the integers that conform the atom. *sign* determines whether the integers are signed or not. The meaning of the other parameters are the same of those in the `Atom` class.

This class has several descendants:

Int8Atom(shape=1, flavor="numarray") Define an atom of type `Int8`.

UInt8Atom(shape=1, flavor="numarray") Define an atom of type `UInt8`.

Int16Atom(shape=1, flavor="numarray") Define an atom of type `Int16`.

UInt16Atom(shape=1, flavor="numarray") Define an atom of type `UInt16`.

Int32Atom(shape=1, flavor="numarray") Define an atom of type `Int32`.

UInt32Atom(shape=1, flavor="numarray") Define an atom of type `UInt32`.

Int64Atom(shape=1, flavor="numarray") Define an atom of type `Int64`.

UInt64Atom(shape=1, flavor="numarray") Define an atom of type `UInt64`.

FloatAtom(shape=1, itemsize=8, flavor="numarray") Define an atom to be of `FloatXX` type, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the floats in the atom and the default is 8 bytes (double precision). The meaning of the other parameters are the same as those in the `Atom` class.

This class has two descendants:

Float32Atom(shape=1, flavor="numarray") Define an atom of type `Float32`.

Float64Atom(shape=1, flavor="numarray") Define an atom of type `Float64`.

ComplexAtom(shape=1, itemsize=16, flavor="numarray") Define an atom to be of `ComplexXX` type, depending on the value of `itemsize`. The `itemsize` parameter sets the number of bytes of the floats in the atom and the default is 16 bytes (double precision complex). The meaning of the other parameters are the same as those in the `Atom` class.

This class has two descendants:

Complex32Atom(shape=1, flavor="numarray") Define an atom of type `Complex32`.

Complex64Atom(shape=1, flavor="numarray") Define an atom of type `Complex64`.

TimeAtom(shape=1, itemsize=8, flavor="numarray") Define an atom to be of type *Time*. Two kinds of time atoms are supported depending on the value of `itemsize`: 4-byte signed integer and 8-byte double precision floating point atoms (the default ones). The meaning of the other parameters are the same as those in the `Atom` class.

Time atoms have a special encoding in the HFD5 file. See appendix A for more information on those types.

This class has two descendants:

Time32Atom(shape=1, flavor="numarray") Define an atom of type `Time32`.

Time64Atom(shape=1, flavor="numarray") Define an atom of type `Time64`.

EnumAtom(enum, dtype='UInt32', shape=1, flavor='numarray') Description of an atom of an enumerated type.

Instances of this class describe the atom type used by an array to store enumerated values. Those values belong to an enumerated type.

The meaning of the `enum` and `dtype` arguments is the same as in `EnumCol` (see 4.16.2). The `shape` and `flavor` arguments have the usual meaning of other `Atom` classes (the `flavor` applies to the representation of concrete read values).

Enumerated atoms also have `stype` and `type` attributes with the same values as in `EnumCol`.

Now, there come two special classes, `ObjectAtom` and `VLString`, that actually do not descend from `Atom`, but which goal is so similar that they should be described here. The difference between them and the `Atom` and descendants classes is that these special classes does not allow multidimensional atoms, nor multiple values per row. A *flavor* can not be specified neither as it is immutable (see below).

Caveat emptor: You are only allowed to use these classes to create `VLArray` objects, not `CArray` and `EArray` objects.

ObjectAtom() This class is meant to fit *any* kind of object in a row of an `VLArray` instance 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 a representant of it per row. However, you can still pass several parameters to the `VLArray.append()` method as they will be regarded as a *tuple* of compound objects (the parameters), so that we still have only one object to be saved in a single row. It does not accept parameters and its flavor is automatically set to `"Object"`, so the reads of rows always returns an arbitrary python object. You can regard `ObjectAtom` types as an easy way to save an arbitrary number of generic python objects in a `VLArray` object.

VLStringAtom() 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 only accepts one object when the base atom is of this type. Besides, it supports Unicode strings (contrarily to `StringAtom`) because it uses the UTF-8 codification (this is why its `atomsize()` method returns always 1) when serializing to disk. It does not accept any parameter and because its *flavor* is automatically set to `"VLString"`, the reads of rows always returns a python string. See the appendix D.3.5 if you are curious on how this is implemented at the low-level. You can regard `VLStringAtom` types as an easy way to save generic variable length strings.

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

4.17 Helper classes

In this section are listed classes that does not fit in any other section and that mainly serve for ancillary purposes.

4.17.1 The `Filters` class

This class is meant to serve as a container that keeps information about the filter properties associated with the enlargeable leaves, that is `Table`, `EArray` and `VLArray` as well as `CArray`.

The public variables of `Filters` are listed below:

complevel The compression level (0 means no compression).

complib The compression filter used (in case of compressed dataset).

shuffle Whether the shuffle filter is active or not.

fletcher32 Whether the fletcher32 filter is active or not.

There are no `Filters` public methods with the exception of the constructor itself that is described next.

```
Filters(complevel=0, complib="zlib", shuffle=1, fletcher32=0)
```

The parameters that can be passed to the `Filters` class constructor are:

complevel Specifies a compress level for data. The allowed range is 0-9. A value of 0 disables compression. The default is that compression is disabled, that balances between compression effort and CPU consumption.

complib Specifies the compression library to be used. Right now, `"zlib"` (default), `"lzo"`, `"ucl"` and `"bzip2"` values are supported. 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 present in the `HDF5` library. This is normally used to improve the compression ratio (at the cost of consuming a little bit more CPU time). A value of 0 disables shuffling and 1 makes it active. The default value depends on whether compression is enabled or not; if compression is enabled, shuffling defaults to be active, else shuffling is disabled.

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 value of 0 disables the checksum and it is the default.

Of course, you can also create an instance and then assign the ones you want to change. For example:

```
import numarray as na
from tables import *

fileh = openFile("test5.h5", mode = "w")
atom = Float32Atom(shape=(0,2))
filters = Filters(complevel=1, complib = "lzo")
filters.fletcher32 = 1
arr = fileh.createEArray(fileh.root, 'earray', atom, "A growable array",
                         filters = filters)
# Append several rows in only one call
arr.append(na.array([[1., 2.],
                     [2., 3.],
                     [3., 4.]], type=na.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(3L, 2), fletcher32, shuffle, lzo(1)) 'A growable array'
  type = Float32
  shape = (3L, 2)
  itemsize = 4
  nrows = 3
  extdim = 0
  flavor = 'numarray'
  byteorder = 'little'
```

4.17.2 The `IndexProps` class

You can use this class to set/unset the properties in the indexing process of a `Table` column. To use it, create an instance, and assign it to the special attribute `_v_indexprops` in a table description class (see 4.16.1) or dictionary.

The public variables of `IndexProps` are listed below:

auto Whether an existing index should be updated or not after a table append operation.

reindex Whether the table columns are to be re-indexed after an invalidating index operation.

filters The filter settings for the different `Table` indexes.

There are no `IndexProps` public methods with the exception of the constructor itself that is described next.

IndexProps(auto=1, reindex=1, filters=None)

The parameters that can be passed to the `IndexProps` class constructor are:

auto Specifies whether an existing index should be updated or not after a table append operation. The default is enable automatic index updates.

reindex Specifies whether the table columns are to be re-indexed after an invalidating index operation (like for example, after a `Table.removeRows` call). The default is to reindex after operations that invalidate indexes.

filters Sets the filter properties for `Column` indexes. It has to be an instance of the `Filters` (see section 4.17.1) class. A `None` value means that the default settings for the `Filters` object are selected.

4.17.3 The `Index` class

This class is used to keep the indexing information for table columns. It is actually a descendant of the `Group` class, with some added functionality.

It has no methods intended for programmer's use, but it has some attributes that maybe interesting for him.

`Index` instance variables

column The column object this index belongs to.

type The type class for the index.

itemsize The size of the atomic items. Specially useful for columns of `CharType` type.

nelements The total number of elements in index.

dirty Whether the index is dirty or not.

filters The `Filters` (see section 4.17.1) instance for this index.

4.17.4 The `Enum` class

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.)

Special methods

__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.

__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.

__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 `__call__()` method.

__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.

__len__() Return the number of enumerated values in the enumerated type.

__iter__() Iterate over the enumerated values.

Enumerated values are returned as `(name, value)` pairs *in no particular order*.

__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).

__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.

... durch planmässiges Tattonieren.
[... through systematic, palpable experimentation.]

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

Chapter 5

Optimization tips

On this chapter, you will get deeper knowledge of `PyTables` internals. `PyTables` has several places where the user can improve the performance of his 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 dataset is small or medium size (say, up to 10 MB), you should not worry about that as the default parameters in `PyTables` are already tuned to handle that perfectly.

5.1 Informing `PyTables` about expected number of rows in tables

The underlying HDF5 library that is used by `PyTables` allows for certain datasets (*chunked* datasets) to take the data in bunches of a certain length, so-called *chunks*, to 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, checksumming, etc. on entire chunks.

An in-memory B-tree 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).

`PyTables` can determine an optimum chunk size to make B-trees adequate to your dataset size if you help it by providing an estimation of the number of rows for a table. This must be made at table creation time by passing this value to the `expectedrows` keyword of the `createTable` method (see 4.2.2).

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

5.2 Accelerating your searches

If you are going to use a lot of searches like the next one:

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

(for future reference, we will call this the *standard* selection mode) and you want to improve the time taken to run it, keep reading.

5.2.1 In-kernel searches

`PyTables` provides a way to accelerate data selections when they are simple, i.e. when only a column is implied in the selection process, through the use of the `where` iterator (see 4.6.2). We will call this mode of

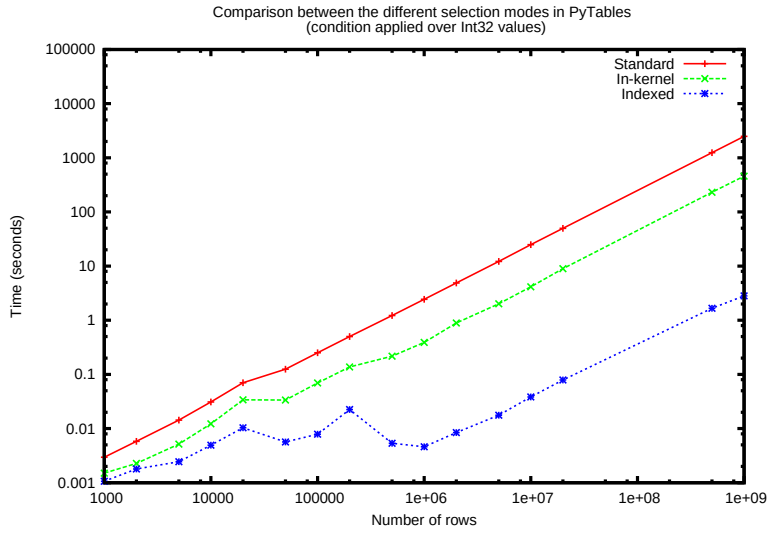


Figure 5.1: Times for different selection modes over `Int32` values. Benchmark made on a machine with Itanium (IA64) @ 900 MHz processors with SCSI disk @ 10K RPM.

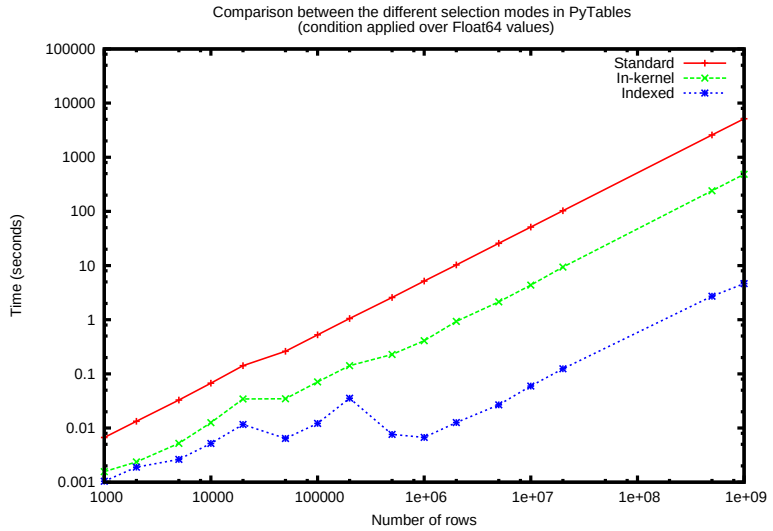


Figure 5.2: Times for different selection modes over `Float64` values. Benchmark made on a machine with Itanium (IA64) @ 900 MHz processors with SCSI disk @ 10K RPM.

selecting data *in-kernel*. Let's see an example of *in-kernel* selection based on the *standard* selection mentioned above:

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

This simple change of mode selection can account for an improvement in search times up to a factor of 10 (see the figure 5.1).

So, where is the trick? It's easy. In the *standard* selection mode the data for column `var1` has to be carried up to Python space so as to evaluate the condition and decide if the `var2` value 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 C speed (with some help of the `numarray` package), so that the only values that are brought to the Python space are the references for `rows` that fulfilled the condition.

You should note, however, that currently the `where` method only accepts conditions along a single column[1]. Fortunately, you can mix the *in-kernel* and *standard* selection modes for evaluating arbitrarily complex conditions along several columns at once. Look at this example:

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

here, we have used a *in-kernel* selection to filter the rows whose `var3` field is equal to string `"foo"`. Then, we apply a *standard* selection to complete the query.

Of course, when you mix the *in-kernel* and *standard* 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.

5.2.2 Indexed searches

When you need more speed than *in-kernel* selections can offer you, `PyTables` offers a third selection method, the so-called *indexed* mode. In this mode, you have to decide which column(s) you are going to do your selections on, and index them. Indexing is just a kind of sort operation, so that next searches along a column will look at the sorted information using a *binary search* which is much faster than a *sequential search*.

You can index your selected columns in several ways:

Declaratively In this mode, you can declare a column as being indexed by passing the *indexed* parameter to the column descriptor. That is:

```
class Example(IsDescription):
    var1 = StringCol(length=4, dflt="", pos=1, indexed=1)
    var2 = BoolCol(0, indexed=1, pos = 2)
    var3 = IntCol(0, indexed=1, pos = 3)
    var4 = FloatCol(0, indexed=0, pos = 4)
```

In this case, we are telling that `var1`, `var2` and `var3` columns will be indexed automatically when you add rows to the table with this description.

Calling Column.createIndex() In this mode, you can create an index even 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, and after doing that, they will behave as regular indexes.

After you have indexed a column, you can proceed to use it through the use of `Table.where` method:

```
row = table
result = [ row['var2'] for row in table.where(table.cols.var1 == "foo") ]
```

or, if you want to add more conditions, you can mix the indexed selection with a standard one:

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

[1] PyTables Pro will address this shortcoming.

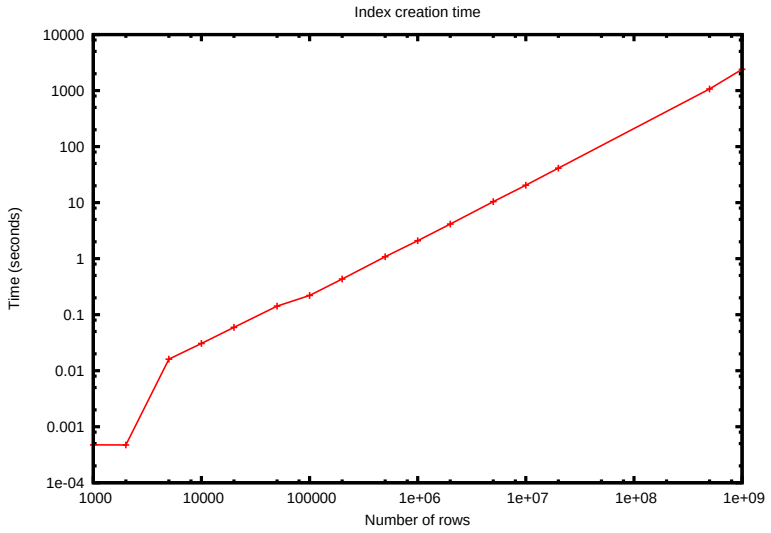


Figure 5.3: Times for indexing a couple of columns of data type `Int32` and `Float64`. Benchmark made on a machine with Itanium (IA64) @ 900 MHz processors with SCSI disk @ 10K RPM.

remember to pass the most restrictive condition to the `where` iterator.

You can see in figures 5.1 and 5.2 that indexing can accelerate quite a lot your data selections in tables. For moderately large tables (> one million rows), you can get speedups in the order of 100x with regard to *in-kernel* selections, and in the order of 1000x with regard to *standard* selections.

One important aspect of indexation in `PyTables` is that it has been implemented with the goal of being capable to manage effectively very large tables. In figure 5.3, you can see that the times to index columns in tables always grow *linearly*. In particular, the time to index a couple of columns with 1 billion of rows each is 40 min. (roughly 20 min. each), which is a quite reasonable figure. This is because `PyTables` has chosen an algorithm that does a *partial* sort of the columns in order to ensure that the indexing time grows *linearly*. On the contrary, most of relational databases try to do a *complete* sort of columns, and this makes the time to index grow much faster with the number of rows.

The fact that relational databases use a complete sorting algorithm for indexes means that their index would be more effective (but not by a large extent) for searching purposes than the `PyTables` approach. However, for relatively large tables (> 10 millions of rows) the time required for completing such a sort can be so large, that indexing is not normally worth the effort. In other words, `PyTables` indexing scales much better than relational databases. So don't worry if you have extremely large columns to index: `PyTables` is designed to cope with that perfectly.

5.3 Compression issues

One of the beauties of `PyTables` is that it supports compression on tables and arrays[2], although it is not used by default. Compression of big amounts of data might be a bit controversial feature, because compression 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.

There is a common scenario where users need to save duplicated data in some record fields, while the others have varying values. In a relational database approach such redundant data can normally be moved to other tables and a relationship between the rows on the separate tables can be created. But that takes analysis and implementation time, and makes the underlying libraries more complex and slower.

`PyTables` transparent compression allows the users to not worry about finding which is their optimum strategy for data tables, but rather use less, not directly related, tables with a larger number of columns while

[2] More precisely, it is supported in `CArray`, `EArray` and `VLArray` objects, but not in `Array` objects.

still not cluttering the database too much with duplicated data (compression is responsible to avoid that). As a side effect, data selections can be made more easily because you have more fields available in a single table, and they can be referred in the same loop. This process may normally end in a simpler, yet powerful manner to process your data (although you should still be careful about in which kind of scenarios the use of compression is convenient or not).

The compression library used by default is the **Zlib** (see Gailly and Adler). 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 *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 Oberhumer) and bzip2 (see Seward). Following the author of LZO (and checked by the author of this section, as you will see soon), LZO offers pretty fast compression (though a small compression ratio) 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 appendix C.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 will probably 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 plot 5.4. 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.

Also, PyTables lets you select different compression levels for Zlib and bzip2, although you may get a bit disappointed by the small improvement that show these compressors when dealing with a combination of numbers and strings as in our example. As a reference, see plot 5.5 for a comparison of the compression achieved by selecting different levels of Zlib. Very oddly, the best compression ratio corresponds to level 1 (!). It's difficult to explain that, but this lesson will serve to reaffirm that there is no replacement for experiments with your own data. In general, it is recommended to select the *lowest* level of compression in order to achieve best performance and decent (if not the best!) compression ratio. See later for more figures on this regard.

Have also a look at graph 5.6. It shows how the speed of writing rows evolves as the size (the row number) of the table grows. Even though in these graphs the size of one single row is 16 bytes, you can most probably extrapolate these figures to other row sizes.

In plot 5.7 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

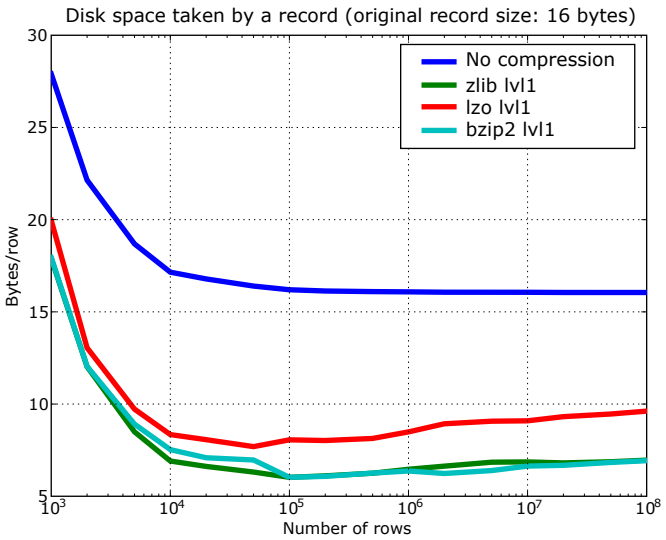


Figure 5.4: Comparison between different compression libraries.

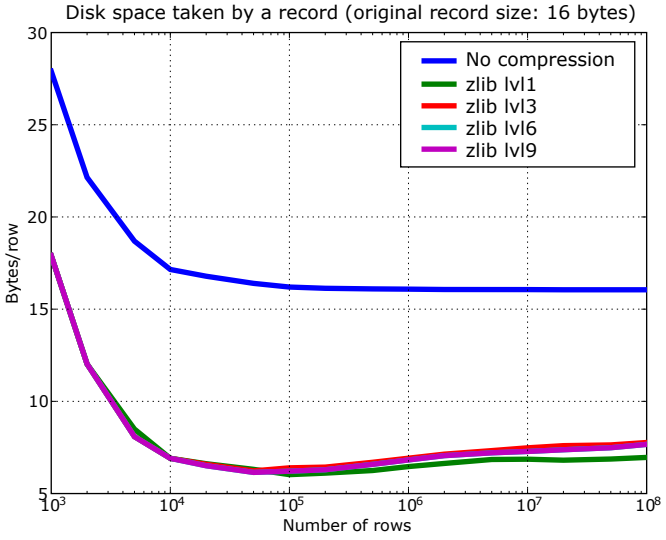


Figure 5.5: Comparison between different compression levels of Zlib.

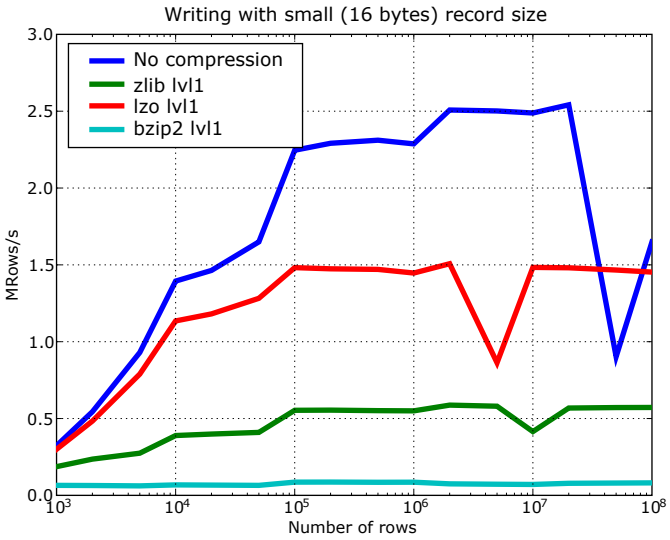


Figure 5.6: Writing tables with several compressors.

LZO does not affect too much the reading performance (and in some points it is actually better), Zlib makes speed to drop to a half, while bzip2 is performing very slow (up to 8x slower).

Also, in the same figure 5.7 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, graph 5.8 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 a 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 above plot is that LZO decompression performance is much better than Zlib, allowing an improvement in overal 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 is not too strange anyway.

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 the shuffle filter 5.4). 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 fairly good compression ratios.

Finally, by looking at the plots 5.9, 5.10, and the aforementioned 5.5 you can see why the recommended compression level to use for all compression libraries is 1. This is the lowest level of compression, but if you take the approach suggested above, the redundant data is to be found normally in the same row, making redundancy locality very high so that a small level of compression should be enough to achieve a good compression ratio on your data tables, saving CPU cycles for doing other things. Nonetheless, in some situations you may want to check for your own how the different compression levels affect your application.

You can select the compression library and level by setting the `complib` and `complevel` keywords in the `Filters` class (see 4.17.1). A compression level of 0 will completely disable compression (the default), 1 is the less CPU time demanding level, while 9 is the maximum level and most 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

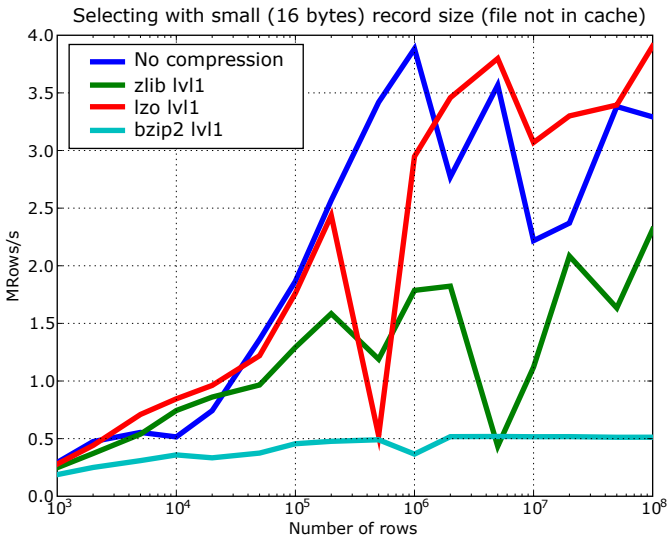


Figure 5.7: Selecting values in tables with several compressors. The file is not in the OS cache.

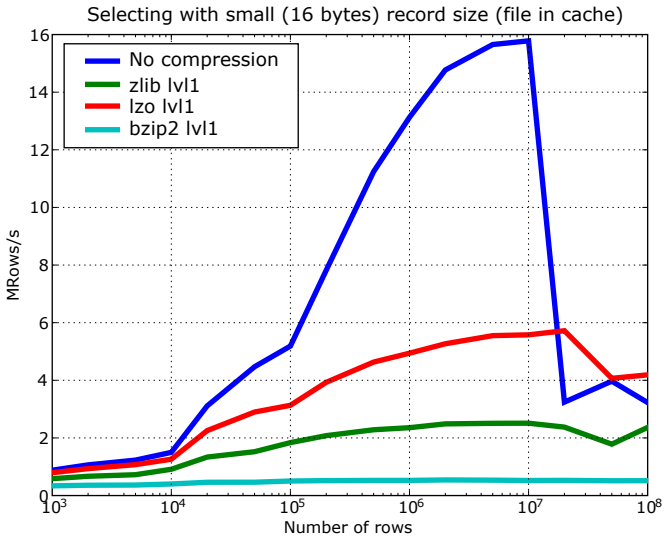


Figure 5.8: Selecting values in tables with several compressors. The file is in the OS cache.

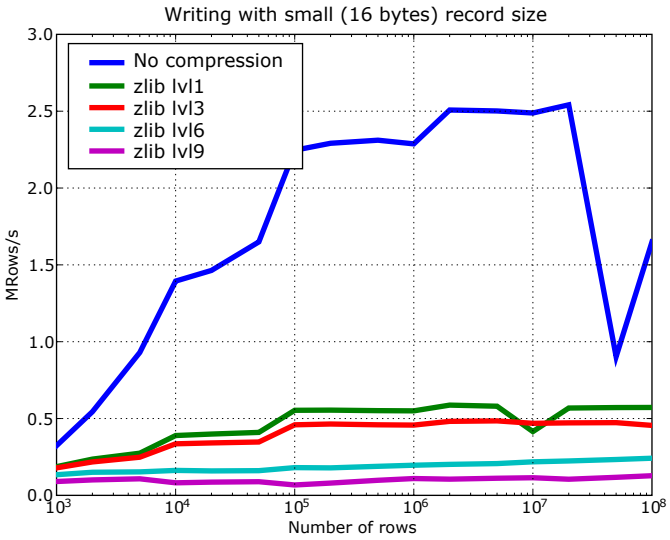


Figure 5.9: Writing in tables with different levels of compression.

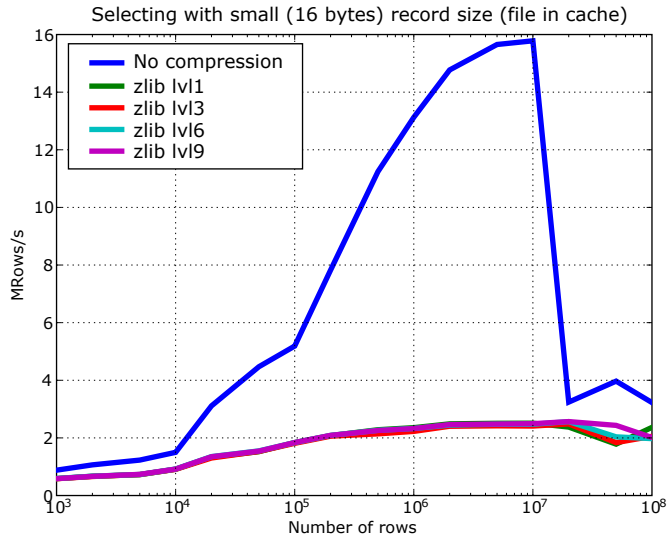


Figure 5.10: Selecting values in tables with different levels of compression. The file is in the OS cache.

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).

5.4 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 (indepen-

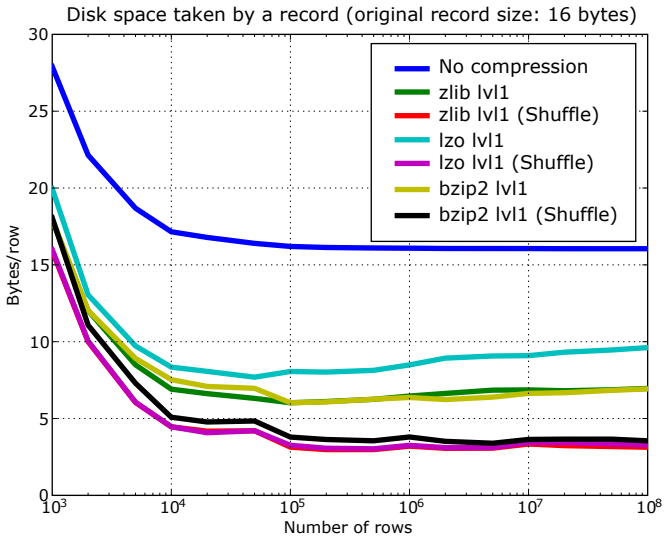


Figure 5.11: Comparison between different compression libraries with and without the *shuffle* filter.

dently of the chosen compressor). It is of course 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 exactly works this mysterious filter? From the HDF5 reference manual: "The shuffle filter deinterlaces 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 graph 5.11 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 to 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[3].

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 5.12, 5.13 and 5.14, 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 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).

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

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

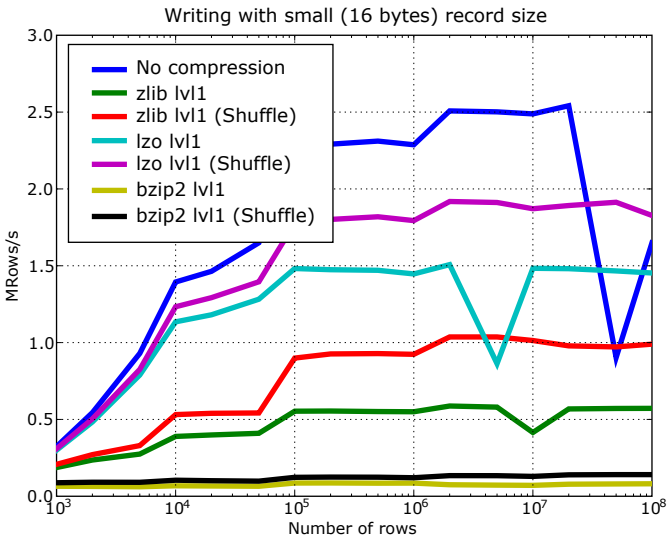


Figure 5.12: Writing with different compression libraries with and without the *shuffle* filter.

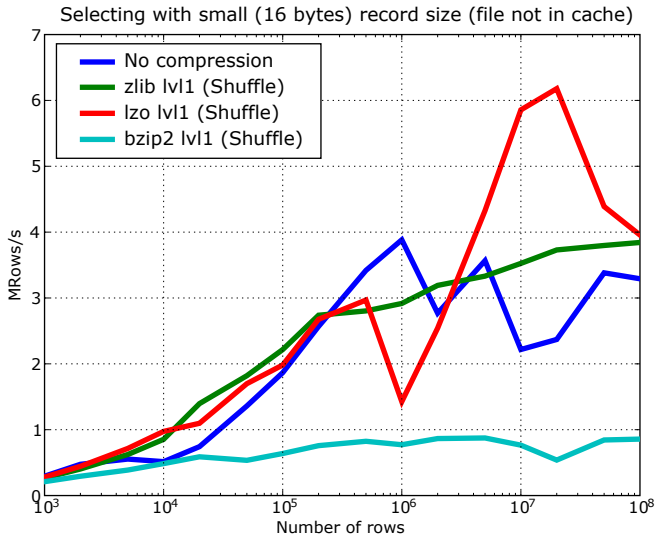


Figure 5.13: Reading with different compression libraries with the *shuffle* filter. The file is not in OS cache.

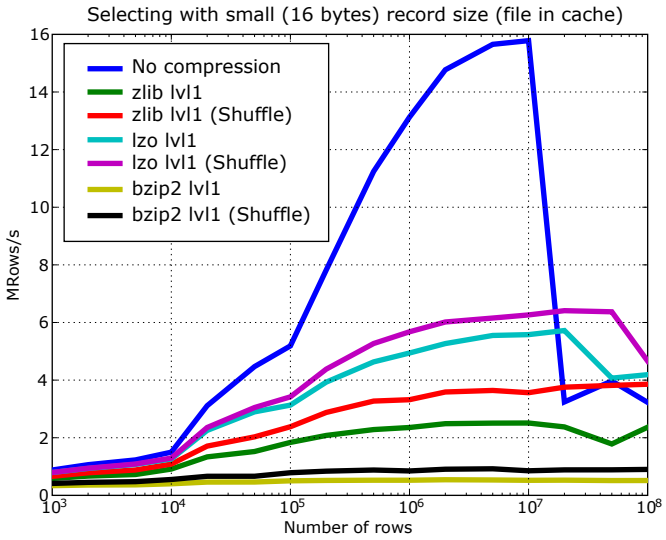


Figure 5.14: Reading with different compression libraries with and without the *shuffle* filter. The file is in OS cache.

(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.

5.5 Using Psyco

Psyco (see Rigo) 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 (at least until Psyco will support yours).

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")
```

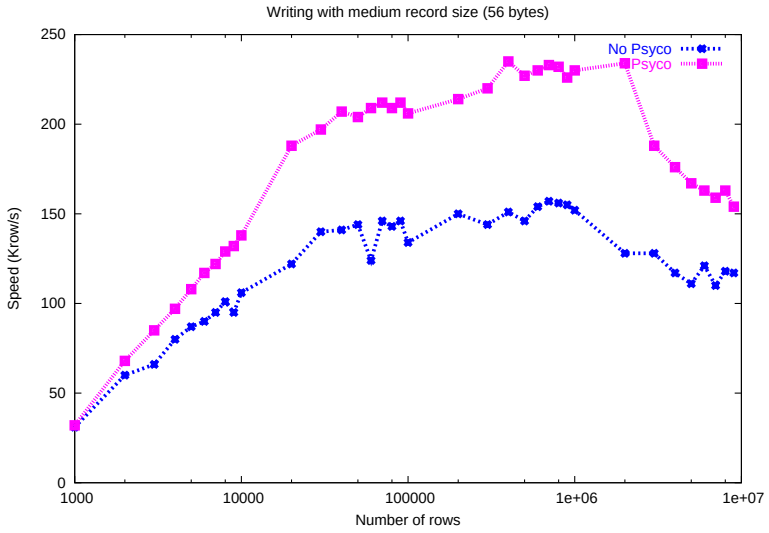


Figure 5.15: Writing tables with/without Psyco.

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 5.15 and 5.16 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.

5.6 Getting the most from the node LRU cache

Starting from PyTables 1.2 on, it has been introduced a new LRU cache that prevents from loading all the nodes of the *object tree* in memory. This cache 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 Altet and Vilata 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 held in memory at any time. As PyTables is meant to be deployed in machines that have potentially low memory, the default for it is quite conservative (you can look at its actual value in the `NODE_CACHE_SIZE` parameter in module `tables/constants.py`). However, if you usually have to deal with files that have much more nodes than the maximum default, and you have a lot of free memory in your system, then you may want to experiment which is the appropriate value of `NODE_CACHE_SIZE` that fits better your needs.

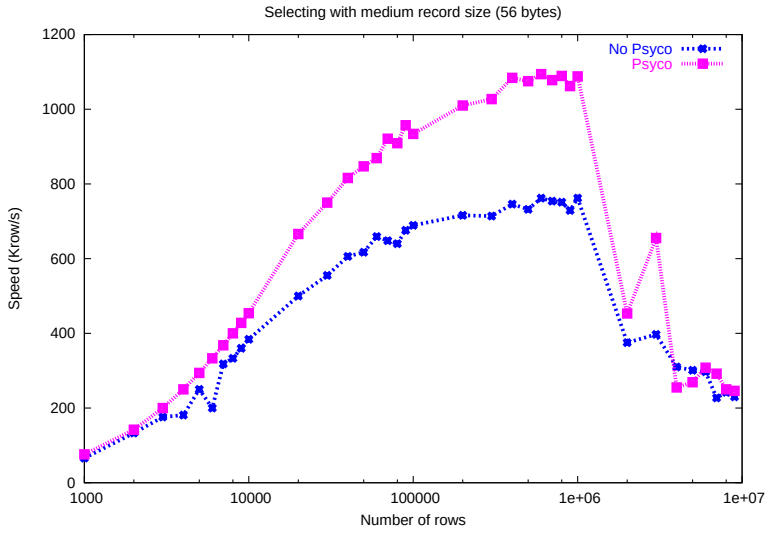


Figure 5.16: Reading tables with/without Psyco.

Table 5.1: Retrieving speed and memory consumption dependency of the number of nodes in LRU cache.

		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
From disk		14	14	1.24	1.24	51	66	1.33	1.31
From cache		14	14	0.53	0.52	65	73	1.35	0.68

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. We will run this small benchmark for different values of the LRU cache size, namely 256 and 1024. You can see the results in table 5.1.

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 does 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, it does not take much more memory than keeping too conservative. On another hand, it might happen that the speed-up that you can achieve by allocating more slots in your cache maybe is not worth the amount of memory used.

Anyway, if you feel that this issue is important for you, setup your own experiments and proceed fine-tuning the `NODE_CACHE_SIZE` parameter.

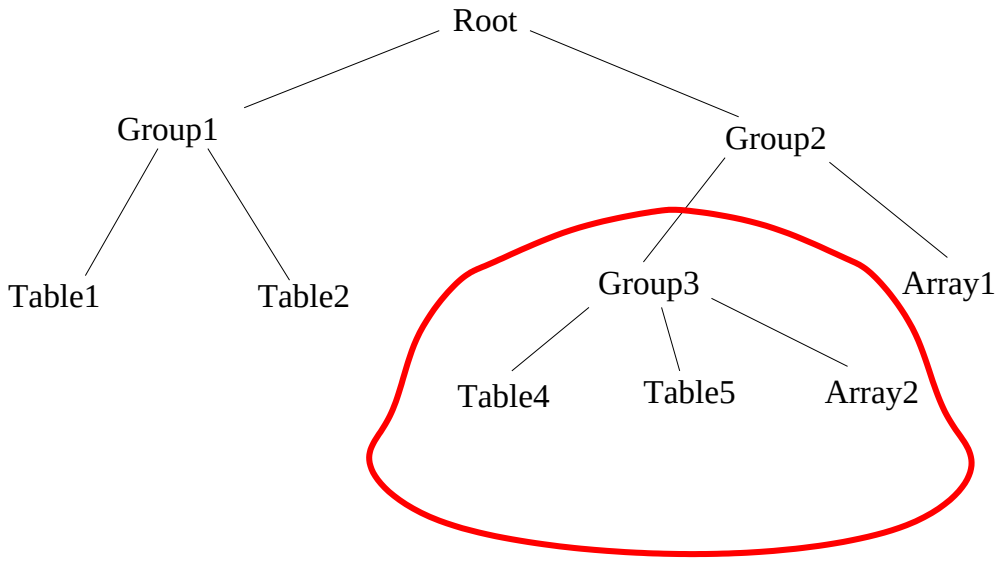


Figure 5.17: Complete tree in file `test.h5`, and subtree of interest for the user.

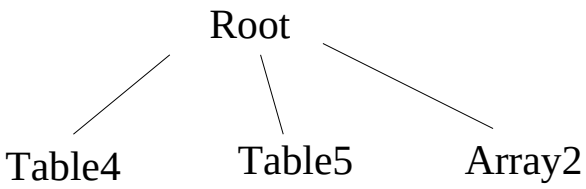


Figure 5.18: Resulting object tree derived from the use of the `rootUEP` parameter.

5.7 Selecting an User Entry Point (UEP) in your tree

Note: After the introduction of the new object tree cache in PyTables 1.2, this feature is not very useful anymore and might become *deprecated* in future versions.

If you have a **huge** tree in your data file with many nodes on it, creating the object tree would take long time. Many times, however, you are interested only in access to a part of the complete tree, so you won't strictly need PyTables to build the entire object tree in-memory, but only the *interesting* part.

This is where the `rootUEP` parameter of `openFile` function (see 4.1.2) can be helpful. Imagine that you have a file called `"test.h5"` with the associated tree that you can see in figure 5.17, and you are interested only in the section marked in red. You can avoid the build of all the object tree by saying to `openFile` that your root will be the `/Group2/Group3` group. That is:

```
fileh = openFile("test.h5", rootUEP="/Group2/Group3")
```

As a result, the actual object tree built will be like the one that can be seen in figure 5.18.

Of course this has been a simple example and the use of the `rootUEP` parameter was not very necessary. But when you have *thousands* of nodes on a tree, you will certainly appreciate the `rootUEP` parameter.

5.8 Compacting your `PyTables` files

Let's suppose that you have a file on which 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 on a table, the space is not automatically recovered on-the-flight. In addition, if you add many more rows to a table than specified in the `expectedrows` keyword in creation time this may affect performance as well, as explained in section 5.1.

In order to cope with these issues, you should be aware that a handy `PyTables` utility called `ptrepack` can be very useful, not only to compact your already existing *leaky* files, but also to adjust some internal parameters (both in memory and in file) in order to create adequate buffer sizes and chunk sizes for optimum I/O speed. Please, check the appendix C.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`, `UCL` 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 NCSA), 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 4.2.2) 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 := 'numarray',
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 `FileNode`s 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 file names are constrained to `PyTables` node names (i.e. most valid Python identifiers). For the moment, if you want arbitrary names you will have to use a translation map (see 4.1.2) or the node title. The same restriction applies to attribute names.

2. Node files can only be opened for read-only or read and append mode. This will be enhanced in the future.

3. There is no universal newline support yet. This is likely to be implemented in a near future.

4. Sparse files (files with lots of zeros) are not treated specially; if you want them to take less space, you should be better off using compression.

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

6.6 `FileNode` module reference

6.6.1 Global constants

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

NetCDF - a `PyTables` NetCDF3 emulation API

7.1 What is NetCDF?

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 Davis *et al.*). A very nice python interface to that library is available in the `Scientific Python NetCDF` module (see Hinsen). 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 Rew *et al.*). The `tables.NetCDF` module does not create HDF5 files that are compatible with netCDF 4 (although this is a long-term goal).

7.2 Using the `tables.NetCDF` module

The module `tables.NetCDF` 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.NetCDF` 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.NetCDF` file. All of the code snippets presented here are included in `examples/netCDF_example.py`. The `tables.NetCDF` module is designed to be used as a drop-in replacement for `Scientific.IO.NetCDF`, with only minor modifications to existing code. The differences between `table.NetCDF` and `Scientific.IO.NetCDF` are summarized in the last section of this chapter.

7.2.1 Creating/Opening/Closing a `tables.NetCDF` file

To create a `tables.netCDF` file from python, you simply call the `NetCDFFile` constructor. This is also the method used to open an existing `tables.netCDF` 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.NetCDF` file is accomplished via the `close` method of `NetCDFFile` object.

Here's an example:

```
>>> import tables.NetCDF 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.NetCDF 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.NetCDF 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.NetCDF file

Most of the data in a `tables.NetCDF` 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.NetCDF.NetCDFVariable instance at 0x244f350>,
 'pressure': <tables.NetCDF.NetCDFVariable instance at 0x244f508>,
 'level': <tables.NetCDF.NetCDFVariable instance at 0x244f0d0>,
 'temp': <tables.NetCDF.NetCDFVariable instance at 0x244f3a0>,
 'time': <tables.NetCDF.NetCDFVariable instance at 0x2564c88>}
```

7.2.4 Attributes in a tables.NetCDF file

There are two types of attributes in a tables.NetCDF 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.NetCDF'
>>> 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.NetCDF
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.NetCDF 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 numarray
>>> levels[:] = numarray.arange(12)+1
>>> latitudes[:] = numarray.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.NetCDF` 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 =  (10L, 12, 90)
>>> import numarray.random_array as 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.NetCDF' ;
        :history = 'Created Wed Nov  9 12:29:13 2005' ;
        :source = 'PyTables Users Guilde' ;
}
```

7.2.6 Efficient compression of tables.NetCDF 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 4.17.1) with complevel=6, complib='zlib' and shuffle=1. 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 numarray.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.NetCDF module - 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 for details).

7.3 tables.NetCDF module 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.NetCDF 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.NetCDF 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 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=1` and `fletcher32=0`.

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.NetCDF` file. The default corresponds to `complevel=6`, `complib='zlib'`, `shuffle=1` and `fletcher32=0`.

h5tonc(filename, packshort=False, scale_factor=None, add_offset=None) Exports the data in a `tables.NetCDF` 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 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.NetCDF files

If Scientific.IO.NetCDF is installed, tables.NetCDF provides facilities for converting between true netCDF version 3 files and tables.NetCDF hdf5 files via the NetCDFFile.h5tonc() and NetCDFFile.nctoh5() class methods. Also, the nctoh5 command-line utility (see Appendix C.3) uses the NetCDFFile.nctoh5() class method.

As an example, look how to convert a tables.NetCDF hdf5 file to a true netCDF version 3 file (named test.nc)

```
>>> scale_factor = {'temp': 1.75e-4}
>>> add_offset = {'temp': 5.}
>>> file.h5tonc('test.nc',packshort=True, \
                scale_factor=scale_factor,add_offset=add_offset)
packing temp as short integers ...
>>> file.close()
```

The dictionaries scale_factor and add_offset are used to optionally pack the data as short integers in the netCDF file. Since netCDF version 3 does not provide automatic compression, packing as short integers is a commonly used way of saving disk space (see this page for more details). The keys of these dictionaries are the variable names to pack, the values are the scale_factors and offsets to use in the packing. The resulting netCDF file will have the scale_factor and add_offset variable attributes set appropriately.

To convert the netCDF file back to a tables.NetCDF 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 =  (10L, 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.NetCDF files have some features not supported in netCDF (such as Complex data types and the ability to make any dimension *unlimited*). tables.NetCDF files which utilize these features cannot be converted to netCDF using NetCDFFile.h5tonc.

7.5 tables.NetCDF file structure

A tables.NetCDF 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.NetCDF module.

7.6 Sharing data in tables.NetCDF files over the internet with OPeNDAP

tables.NetCDF 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.NetCDF 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.NetCDF files must have the extension .h5 or .hdf5. Unfortunately, tables.NetCDF 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.NetCDF datasets with OPeNDAP.

7.7 Differences between the Scientific.IO.NetCDF API and the tables.NetCDF API

1. tables.NetCDF 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.NetCDF 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.NetCDF, 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.NetCDF 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 numarray.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.NetCDF, 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.NetCDF 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.NetCDF 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=1` and `fletcher32=0`.

6. `tables.NetCDF` 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.NetCDF` file using the `nctoh5` class method.

7. In `tables.NetCDF` 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.NetCDF` 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.NetCDF NetCDFFile` instance.

Part III

Appendixes

Appendix A

Supported data types in PyTables

The Table, Array, CArray, VLArray and EArray classes can all handle the complete set of data types supported by the numarray package (see Greenfield *et al.*), NumPy (see Oliphant *et al.*) and Numeric (see Ascher *et al.*) in Python. The data types for table fields can be set via the constructor for the Col class and its descendants (see 4.16.2) while array elements can be set through the use of the Atom class and its descendants (see 4.16.3).

In addition to those data types, PyTables' Table, VLArray and EArray classes do support some *aliasing* data types for their columns and atoms. Each one of these aliasing types corresponds to one numarray type, but they also have special meanings for PyTables. They can be seen as the ordinary types they are associated with, plus some additional meaning. Since they do not exist as numarray types, they can only be specified to PyTables using strings.

Currently, the only supported aliasing data type is *Time*. Two kinds of time values can be handled: 4-byte signed integer and 8-byte double precision floating point. Both of them reflect the number of seconds since the Unix Epoch, i.e. Jan 1 00:00:00 UTC 1970. Their types correspond to numarray's Int32 and Float64, respectively. Time values are stored in the HDF5 file using the H5T_TIME class. Integer times are stored as is, 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 numarray representation of an enumerated value depends on the concrete base type used to store the enumeration in the HDF5 file. Enumerations are similar to aliasing data types in the sense that enumerated data is handled as regular numarray data. Enumerations are also specified to PyTables using a string type, with an additional Enum (see 4.17.4) instance.

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.

A quick reference to the complete set of data types supported by PyTables is given in table A.

Table A.1: Data types supported for array elements and tables columns in PyTables.

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
Bool	boolean	unsigned char	1	Boolean
Int8	8-bit integer	signed char	1	Integer
UInt8	8-bit unsigned integer	unsigned char	1	Integer
Int16	16-bit integer	short	2	Integer
UInt16	16-bit unsigned integer	unsigned short	2	Integer
Int32	integer	int	4	Integer
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
Complex32	single-precision complex	struct {float r, i;}	8	Complex
Complex64	double-precision complex	struct {double r, i;}	16	Complex
CharType	arbitrary length string	char[]	*	String
Time32	integer time	POSIX's time_t	4	Integer
Time64	floating point time	POSIX's struct timeval	8	Float
Enum	enumerated value	enum	-	-

Appendix B

Using nested record arrays

B.1 Introduction

Nested record arrays are a generalization of the record array concept. 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.

Each nested record array is a `NestedRecArray` object in the `tables.nestedrecords` module. 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.nestedrecords` module. 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.nestedrecords 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.nestedrecords 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, makeFormats, makeNames and makeDescr, that can be useful to obtain the structure specification to be used with 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.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')]
```

B.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 `'/'`, for instance:

```
>>> print nra2.field('info/name')
['a1', 'a2']
>>>
```

Eventually, the top level fields of the nested recarray can be accessed passing an integer argument to the `field()` method:

```
>>> print nra2.field(1)
[[ 0.5 1. ] [ 0.  0. ]]
>>>
```

An alternative to the `field()` method is the use of the `fields` attribute. It is intended mainly for interactive usage in the Python console. For example:

```
>>> nra2.fields.id
[1, 2]
>>> nra2.fields.info.fields.name
['a1', 'a2']
>>>
```

Rows of nested recarrays can be read using the typical index syntax. The rows are retrieved as `NestedRecord` objects:

```
>>> print nra2[0]
(1L, array([ 0.5,  1. ], type=Float32), ('a1', 1j))
>>>
>>> nra2[0].__class__
<class tables.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.

B.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 C

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.

C.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*).

C.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 [-R start,stop,step] [-a] [-h] [-d] [-v] file[:nodepath]
  -R RANGE -- Select a RANGE of rows in the form "start,stop,step"
  -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)
  -d -- Dump data information on leaves
  -h -- Print help on usage
  -v -- Dump more meta-information on nodes
```

C.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
Filename: 'vlarray1.h5' Title: '' , Last modif.: 'Fri Feb  6 19:33:28 2004' ,
 rootUEP='/', filters=Filters(), Format version: 1.2
```

```
/ (Group) ''
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
```

we can see that the file contains a 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
/ (Group) ''
  children := ['vlarray1' (VLArray)]
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Atom(type=Int32, shape=1, flavor='Numeric')
  nrows = 4
  flavor = 'Numeric'
  byteorder = 'little'
```

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
/ (Group) ''
  children := ['vlarray1' (VLArray)]
  /._v_attrs (AttributeSet), 5 attributes:
   [CLASS := 'GROUP',
    FILTERS := None,
    PYTABLES_FORMAT_VERSION := '1.2',
    TITLE := '',
    VERSION := '1.0']
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Atom(type=Int32, shape=1, flavor='Numeric')
  nrows = 4
  flavor = 'Numeric'
  byteorder = 'little'
  /vlarray1.attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'Numeric',
    TITLE := 'ragged array of ints',
    VERSION := '1.0']
```

Let's have a look at the real data:

```
$ ptdump -d vlarray1.h5
/ (Group) ''
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[array([5, 6]), array([5, 6, 7]), array([5, 6, 9, 8]), array([ 5,  6,  9, 10, 12])]
```

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 -R2,4 -d vlarray1.h5:/vlarray1
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[array([5, 6, 9, 8]), array([ 5,  6,  9, 10, 12])]
```

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 -R2,4 -vad vlarray1.h5:/vlarray1
/vlarray1 (VLArray(4,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Atom(type=Int32, shape=1, flavor='Numeric')
  nrows = 4
  flavor = 'Numeric'
  byteorder = 'little'
  /vlarray1.attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'Numeric',
    TITLE := 'ragged array of ints',
    VERSION := '1.0']
  Data dump:
[array([5, 6, 9, 8]), array([ 5,  6,  9, 10, 12])]
```

C.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.

C.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]
  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.
 --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", "ucl" 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.
```

C.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
Filename: 'tutorial1.h5' Title: 'Test file' , Last modif.: 'Fri Feb  6
  19:33:28 2004' , rootUEP='/', filters=Filters(), Format version: 1.2
/ (Group) '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(10L,)) '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
Filename: 'reduced.h5' Title: '' , Last modif.: 'Fri Feb 20 15:26:47 2004' ,
 rootUEP='/', filters=Filters(), Format version: 1.2
/ (Group) ''
/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
Filename: 'reduced.h5' Title: '' , Last modif.: 'Fri Feb 20 15:39:15 2004' ,
 rootUEP='/', filters=Filters(), Format version: 1.2
```

```
/ (Group) ''
/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
Filename: 'reduced.h5' Title: '' , Last modif.: 'Fri Feb 20 15:41:57 2004' ,
 rootUEP='/', filters=Filters(), Format version: 1.2
/ (Group) ''
/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
Filename: 'reduced.h5' Title: '' , Last modif.: 'Fri Feb 20 15:52:22 2004',
 rootUEP='/', filters=Filters(), Format version: 1.2
/ (Group) ''
/rawdata (Table(10L,)) '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 --> exceptions.ValueError: The destination
 (/rawdata (Table(10L,)) 'Readout example') already exists.
 Assert the overwrite parameter if you really want to overwrite it.
The destination file looks like:
Filename: 'reduced.h5' Title: ''; Last modif.: 'Fri Feb 20 15:52:22 2004';
 rootUEP='/'; filters=Filters(), Format version: 1.2
/ (Group) ''
/rawdata (Table(10L,)) '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 358, in ?
    start=start, stop=stop, step=step)
  File "../utils/ptrepack", line 111, 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 `/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
Filename: 'reduced.h5' Title: ''; Last modif.: 'Fri Feb 20 16:02:20 2004';
 rootUEP='/'; filters=Filters(), Format version: 1.2
/ (Group) ''
/rawdata (Table(10L,), 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
Filename: 'reduced.h5' Title: ''; Last modif.: 'Fri Feb 20 16:17:13 2004';
 rootUEP='/'; filters=Filters(); Format version: 1.2
/ (Group) ''
/rawdata (Table(10L,), 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(3L,)) '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.

C.3 nctoh5

This tool is able to convert a file in `NetCDF` format to a `PyTables` file (and hence, to a HDF5 file). However, for this to work, you will need the NetCDF interface for Python that comes with the excellent `Scientific Python` (see Hinsen) 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 Jones *et al.*) 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!

C.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)] [--unpackshort=(0|1)]
 [--quantize=(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", "ucl" 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).
 --unpackshort=(0|1) -- unpack short integer variables to float variables
     using scale_factor and add_offset netCDF variable attributes
     (not active by default).
 --quantize=(0|1) -- quantize data to improve compression using
     least_significant_digit netCDF variable attribute (not active by default).
     See http://www.cdc.noaa.gov/cdc/conventions/cdc_netcdf_standard.shtml
     for further explanation of what this attribute means.
```

If you have followed the small tutorial on the `ptrepack` utility (see C.2), you should easily realize what most of the different flags would mean.

Appendix D

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 **1.6 version of PyTables file format** (introduced in PyTables version 1.3). At this stage, this file format is considered stable enough to do not introduce significant changes during a reasonable amount of time. As time goes by, some changes will 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).

D.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 '1.6' 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'.

D.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'.

D.3 Mandatory attributes, storage layout and supported data types for Leaves

This depends on the kind of Leaf. The format for each type follows.

D.3.1 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'.

FLAVOR This is meant to provide the information about the kind of object kept in the Table, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one the next string values:

"numarray" The read operations will return a numarray object.

"numpy" The read operations will be return as a NumPy object.

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_D32BE 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_D64BE 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 of 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.

Currently, PyTables does not support nested H5T_COMPOUND types, the only exception being supporting complex numbers in Table objects as described above.

D.3.2 Array format

Mandatory attributes

The next attributes are mandatory for *array* structures:

CLASS Must be set to 'ARRAY'.

FLAVOR This is meant to provide the information about the kind of object kept in the Array, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one the next string values:

"numarray" The read operations will return a numarray object.

"numpy" The read operations will return a NumPy object.

"numeric" The read operations will return a Numeric object.

"python" The read operations will return a Python list object in case the dataset has dimensionality. If the dataset is an scalar, then an appropriate Python scalar will be returned instead.

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 D.3.1 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 D.3.1 for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed in Array objects.

D.3.3 CArray format

Mandatory attributes

The next attributes are mandatory for *carray* structures:

CLASS Must be set to 'CARRAY'.

FLAVOR This is meant to provide the information about the kind of objects kept in the CArray, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take the same values as the Array object.

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 D.3.1 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 D.3.1 for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed in Array objects.

D.3.4 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 extensible dimension. Only one extensible dimension is supported right now.

FLAVOR This is meant to provide the information about the kind of objects kept in the EArray, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take the same values as the Array object (see D.3.2), except "Int" and "Float".

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 D.3.1 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 D.3.1.

You should note that H5T_ARRAY class data types are not allowed in EArray objects.

D.3.5 VLArray format

Mandatory attributes

The next attributes are mandatory for *vlarray* structures:

CLASS Must be set to 'VLARRAY'.

FLAVOR This is meant to provide the information about the kind of objects kept in the VLArray, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one of the next values:

"numarray" The dataset will be returned as a numarray object.

"numpy" The dataset will be returned as a NumPy object.

"numeric" The dataset will be returned as an Numeric object.

"python" The dataset will be returned as a Python `List` object in case the dataset has dimensionality. If the dataset is an scalar, then an appropriate Python scalar will be returned instead.

"Object" The elements in the dataset will be interpreted as pickled (i.e. serialized objects through the use of the `Pickle` Python module) objects and returned as Python *generic* objects. Only one of such objects will be deserialized per entry. As the `Pickle` module is not normally available in other languages, this flavor won't be useful in general.

"VLString" The elements in the dataset will be returned as Python `String` objects of *any* length, with the twist that **Unicode** strings are supported as well (provided you use the **UTF-8** codification, see below). However, only one of such objects will be deserialized per entry.

TITLE A string where the user can put some description on what is this dataset used for.

VERSION Should contain the string `'1.2'`.

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 D.3.1. 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 D.3.1 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` special flavors, the base for the VL datatype is always a H5T_NATIVE_UCHAR. 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.

In addition, if you plan to use a `VLString` flavor for your text data and you are using ascii-7 (7 bits ASCII) codification for your strings, but you don't know (or just don't want) to convert it to the required UTF-8 codification, you should not worry too much about that because the ASCII characters with values in the range [0x00, 0x7f] are directly mapped to Unicode characters in the range [U+0000, U+007F] and the UTF-8 encoding has the useful property that an UTF-8 encoded ascii-7 string is indistinguishable from a traditional ascii-7 string. So, you will not need any further conversion in order to save your ascii-7 strings and have an `VLString` flavor.

Bibliography

ALTET, Francesc 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.
URL http://pytables.sourceforge.net/doc/NewObjectTreeCache.pdf 5, 113

ASCHER, David, Paul F. DUBOIS, Konrad HINSEN, Jim HUGUNIN, and Travis OLIPHANT, : *Numerical Python*. Package to speed-up arithmetic operations on arrays of numbers.
URL http://sourceforge.net/projects/numpy/ 4, 9, 139

CÁRABOS, Coop. V., : *Vitables. A GUI for PyTables/HDF5 files*. It is a graphical tool for browsing and editing files in both PyTables and HDF5, formats.
URL http://www.carabos.com/products/vitables.html 6

DAVIS, Glenn, 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.
URL http://www.unidata.ucar.edu/packages/netcdf/ 127

EWING, Greg, : *Pyrex. A Language for Writing Python Extension Modules*.
URL http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex 9

GAILLY, JeanLoup and Mark ADLER, : *zlib. A Massively Spiffy Yet Delicately Unobtrusive Compression Library*. A standard library for compression purposes.
URL http://www.gzip.org/zlib/ 9, 105

GREENFIELD, Perry, Todd MILLER, Richard L. WHITE, *et al.*, : *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.
URL http://stsdas.stsci.edu/numarray/ 4, 9, 82, 139

HINSEN, Konrad, : *Scientific Python*. Collection of Python modules useful for scientific computing.
URL http://starship.python.net/~hinsen/ScientificPython/ 127, 150

JONES, Eric, Travis OLIPHANT, Pearu PETERSON, *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.
URL http://www.scipy.org 150

MERTZ, David, : *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.
URL http://www-106.ibm.com/developerworks/xml/library/xml-matters2/index.html 5

NCSA, : *What is HDF5?* Concise description about HDF5 capabilities and its differences from earlier versions (HDF4).
URL http://hdf.ncsa.uiuc.edu/whatishdf5.html 3, 119

OBERHUMER, Markus F.X.J., : *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.
URL http://www.oberhumer.com/opensource/ 9, 105

OLIPHANT, Travis *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, is more efficient as well.
URL http://numeric.scipy.org/ 4, 9, 139

REW, Russ, Mike FOLK, *et al.*, : *Netcdf-4. Network Common Data Form version 4*. Merging the NetCDF and HDF5 Libraries.
URL http://www.unidata.ucar.edu/software/netcdf/netcdf-4/ 127

RIGO, Armin, : *Psyco. A Python specializing compiler*. Run existing Python software faster, with no change in your source.
URL http://psyco.sourceforge.net 112

SEWARD, Julian, : *bzip2. A high performance lossless compressor*. It offers very high compression ratios within reasonable times.
URL http://www.bzip.org/ 9, 105

WILKE, Alexis, 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.
URL http://gnuwin32.sourceforge.net/ 11, 12