Hierarchical datasets in Python

PyTables User Guide

Release 3.2.0

PyTables maintainers

May 06, 2015

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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

Date May 06, 2015

Version 3.2.0

Home Page http://www.pytables.org

Copyright Notice and Statement for PyTables User's Guide

Part I

The PyTables Core Library

CHAPTER

ONE

INTRODUCTION

> La sabiduría no vale la pena si no es posible servirse de ella para inventar una nueva manera de preparar los garbanzos.
>
> [Wisdom isn't worth anything if you can't use it to come up with a new way to cook garbanzos.]
>
> —Gabriel García Márquez, A wise Catalan in *"Cien años de soledad"*

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

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

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

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

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

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

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

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

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

The next section describes the most interesting capabilities of PyTables.

1.1 Main Features

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

- *Support for table entities:* You can tailor your data adding or deleting records in your tables. Large numbers of rows (up to 2**63, much more than will fit into memory) are supported as well.
- *Multidimensional and nested table cells:* You can declare a column to consist of values having any number of dimensions besides scalars, which is the only dimensionality allowed by the majority of relational databases. You can even declare columns that are made of other columns (of different types).
- *Indexing support for columns of tables:* Very useful if you have large tables and you want to quickly look up for values in columns satisfying some criteria.
- *Support for numerical arrays:* NumPy (see *[NUMPY]*) arrays can be used as a useful complement of tables to store homogeneous data.
- *Enlargeable arrays:* You can add new elements to existing arrays on disk in any dimension you want (but only one). Besides, you are able to access just a slice of your datasets by using the powerful extended slicing mechanism, without need to load all your complete dataset in memory.
- *Variable length arrays:* The number of elements in these arrays can vary from row to row. This provides a lot of flexibility when dealing with complex data.
- *Supports a hierarchical data model:* Allows the user to clearly structure all data. PyTables builds up an *object tree* in memory that replicates the underlying file data structure. Access to objects in the file is achieved by walking through and manipulating this object tree. Besides, this object tree is built in a lazy way, for efficiency purposes.
- *User defined metadata:* Besides supporting system metadata (like the number of rows of a table, shape, flavor, etc.) the user may specify arbitrary metadata (as for example, room temperature, or protocol for IP traffic that was collected) that complement the meaning of actual data.
- *Ability to read/modify generic HDF5 files:* PyTables can access a wide range of objects in generic HDF5 files, like compound type datasets (that can be mapped to Table objects), homogeneous datasets (that can be mapped to Array objects) or variable length record datasets (that can be mapped to VLArray objects). Besides, if a dataset is not supported, it will be mapped to a special UnImplemented class (see *The UnImplemented class*), that will let the user see that the data is there, although it will be unreachable (still, you will be able to access the attributes and some metadata in the dataset). With that, PyTables probably can access and *modify* most of the HDF5 files out there.
- *Data compression:* Supports data compression (using the *Zlib*, *LZO*, *bzip2* and *Blosc* compression libraries) out of the box. This is important when you have repetitive data patterns and don't want to spend time searching for an optimized way to store them (saving you time spent analyzing your data organization).
- *High performance I/O:* On modern systems storing large amounts of data, tables and array objects can be read and written at a speed only limited by the performance of the underlying I/O subsystem. Moreover, if your data is compressible, even that limit is surmountable!
- *Support of files bigger than 2 GB:* PyTables automatically inherits this capability from the underlying HDF5 library (assuming your platform supports the C long long integer, or, on Windows, __int64).

- *Architecture-independent:* PyTables has been carefully coded (as HDF5 itself) with little-endian/big-endian byte ordering issues in mind. So, you can write a file on a big-endian machine (like a Sparc or MIPS) and read it on other little-endian machine (like an Intel or Alpha) without problems. In addition, it has been tested successfully with 64 bit platforms (Intel-64, AMD-64, PowerPC-G5, MIPS, UltraSparc) using code generated with 64 bit aware compilers.

1.2 The Object Tree

The hierarchical model of the underlying HDF5 library allows PyTables to manage tables and arrays in a tree-like structure. In order to achieve this, an *object tree* entity is *dynamically* created imitating the HDF5 structure on disk. The HDF5 objects are read by walking through this object tree. You can get a good picture of what kind of data is kept in the object by examining the *metadata* nodes.

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

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

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

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

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

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

```
from tables import *

class Particle(IsDescription):
```

[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]*).

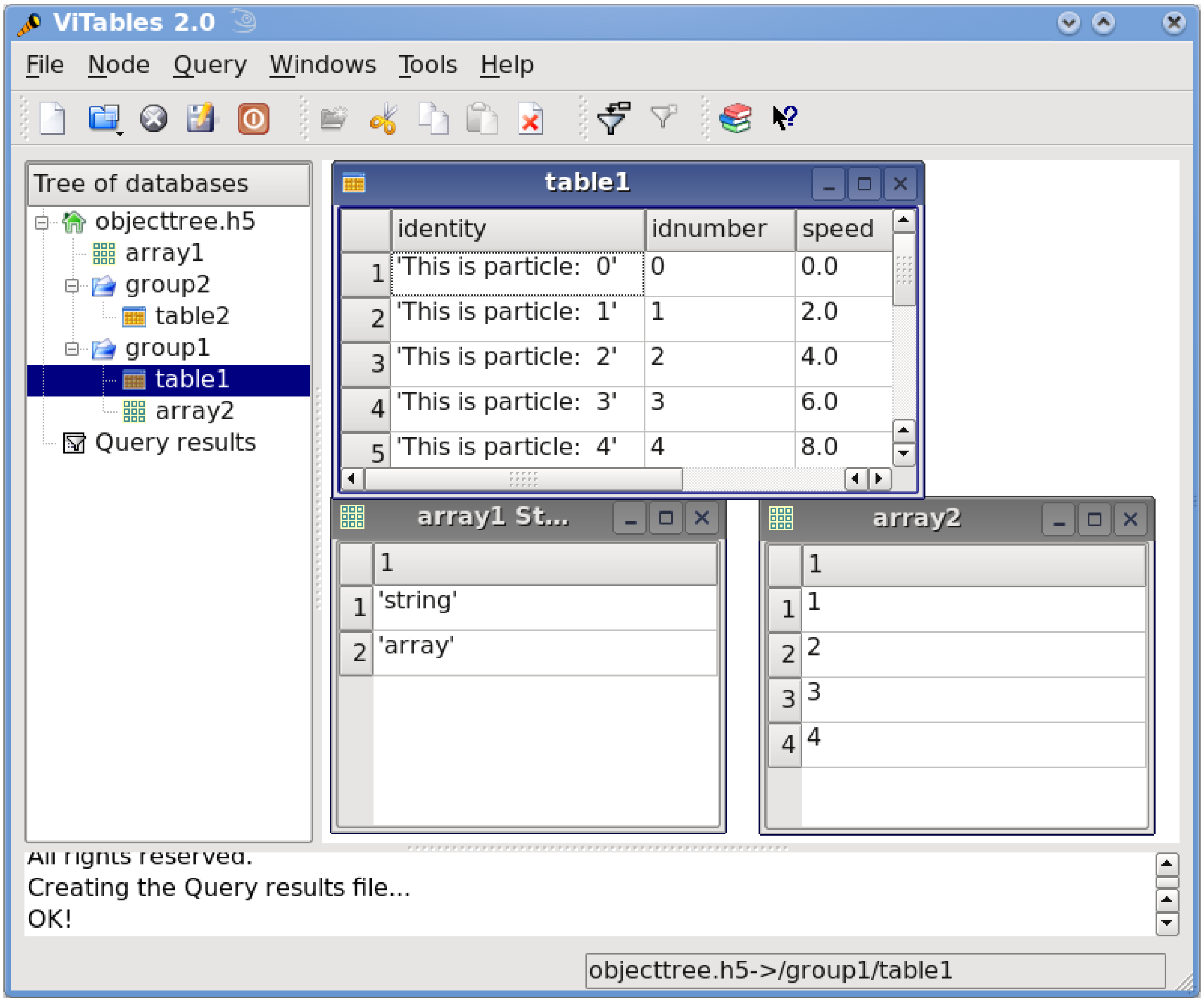
```
    identity = StringCol(itemsize=22, dflt=" ", pos=0)  # character String
    idnumber = Int16Col(dflt=1, pos = 1)  # short integer
    speed    = Float32Col(dflt=1, pos = 2)  # single-precision

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

# Get the HDF5 root group
root = fileh.root

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

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

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

# Create the last table in group2
array2 = fileh.create_array("/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* [3]. When the file is created, the metadata in the object tree is updated in memory while the actual data is saved to disk. When you close the file the object tree is no longer available. However, when you reopen this file the object tree will be reconstructed in memory from the metadata on disk (this is done in a lazy way, in order to load only the objects that are required by the user), allowing you to work with it in exactly the same way as when you originally created it.

In *Figure2*, you can see an example of the object tree created when the above objecttree.h5 file is read (in fact, such an object tree is always created when reading any supported generic HDF5 file). It is worthwhile to take your time to understand it [4]. It will help you understand the relationships of in-memory PyTables objects.

[3] We have used ViTables (see *[VITABLES]*) 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.



	identity	idnumber	speed
1	'This is particle: 0'	0	0.0
2	'This is particle: 1'	1	2.0
3	'This is particle: 2'	2	4.0
4	'This is particle: 3'	3	6.0
5	'This is particle: 4'	4	8.0

	1
1	'string'
2	'array'

	1
1	1
2	2
3	3
4	4

```
All rights reserved.
Creating the Query results file...
OK!
```

objecttree.h5->/group1/table1

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

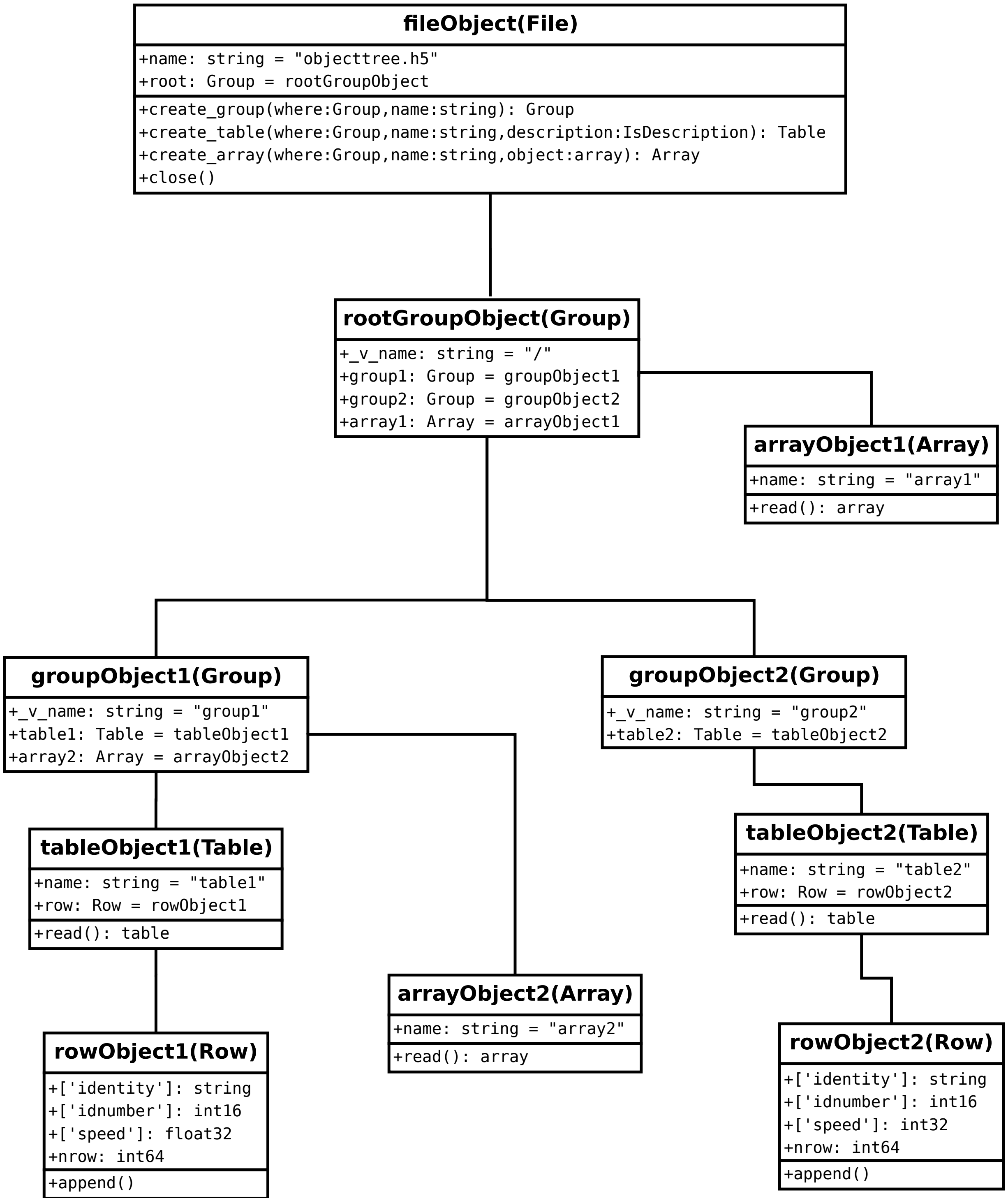


Fig. 1.2: **Figure 2: A PyTables object tree example.**

CHAPTER

TWO

INSTALLATION

> Make things as simple as possible, but not any simpler.
>
> —Albert Einstein

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

However, if you want to go straight to binaries that 'just work' for the main platforms (Linux, Mac OSX and Windows), you might want to use the excellent Anaconda or Canopy distributions. PyTables usually distributes its own Windows binaries too; go *Binary installation (Windows)* for instructions. Finally Christoph Gohlke also maintains an excellent suite of a variety of binary packages for Windows at his site.

2.1 Installation from source

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

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

To compile PyTables you will need a recent version of Python, the HDF5 (C flavor) library from http://www.hdfgroup.org, and the NumPy (see *[NUMPY]*) and Numexpr (see *[NUMEXPR]*) packages.

2.1.1 Prerequisites

First, make sure that you have

- Python >= 2.6 including Python 3.x (Python >= 2.7 is highly recommended)
- HDF5 >= 1.8.4 (>=1.8.7 is strongly recommended)
- NumPy >= 1.7.1
- Numexpr >= 2.4
- Cython >= 0.14
- c-blosc >= 1.4.1 (it is bundled with PyTables sources but the user can use an external version of sources using the `BLOSC_DIR` environment variable or the `--blosc` flag of the `setup.py`)
- argparse (only Python 2.6, it is used by the **`pt2to3`** utility)
- unittest2 (only Python 2.6)

installed (for testing purposes, we are using HDF5 1.8.12, NumPy 1.8.0 and Numexpr 2.4.1 currently). If you don't, fetch and install them before proceeding.

Note: Currently PyTables does not use setuptools by default so do not expect that the setup.py script automatically install all packages PyTables depends on.

Compile and install these packages (but see *Windows prerequisites* for instructions on how to install pre-compiled binaries if you are not willing to compile the prerequisites on Windows systems).

For compression (and possibly improved performance), you will need to install the Zlib (see *[ZLIB]*), which is also required by HDF5 as well. You may also optionally install the excellent LZO compression library (see *[LZO]* and *Compression issues*). The high-performance bzip2 compression library can also be used with PyTables (see *[BZIP2]*).

The Blosc (see *[BLOSC]*) compression library is embedded in PyTables, so this will be used in case it is not found in the system. So, in case the installer warns about not finding it, do not worry too much ;)

Unix

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

```
--hdf5=/stuff/hdf5-1.8.12
--lzo=/stuff/lzo-2.02
--bzip2=/stuff/bzip2-1.0.5
--blosc=/stuff/blosc-1.3.2
```

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

```
--lflags="-Xlinker -rpath -Xlinker /stuff/hdf5-1.8.12/lib"
```

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

```
LIBS="hdf5-1.8.12 nsl"
```

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

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

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

Hint: some GNU/Linux distributions provide a packaged version of the HDF5 libraries with MPI support. In this case you may need to specify the path of the MPH headers as additional include directory.

On Ubuntu 12.04 the following command has been reported to work:

```
$ C_INCLUDE_PATH=/usr/lib/openmpi/include pip install --upgrade tables
```

Starting from PyTables 3.2 can also query the *pkg-config* database to find the required packages. If available, pkg-config is used by default unless explicitly disabled.

To suppress the use of *pkg-config*:

```
$ python setup.py build --use-pkgconfig=FALSE
```

or use the `USE-PKGCONFIG` environment variable:

```
$ env USE_PKGCONFIG=FALSE python setup.py build
```

Windows

You can get ready-to-use Windows binaries and other development files for most of the following libraries from the GnuWin32 project (see *[GNUWIN32]*). In case you cannot find the LZO binaries in the GnuWin32 repository, you can find them at http://sourceforge.net/projects/pytables/files/lzo-win. Once you have installed the prerequisites, setup.py needs to know where the necessary library *stub* (.lib) and *header* (.h) files are installed. You can set the path to the include and dll directories for the HDF5 (mandatory) and LZO, BZIP2, BLOSC (optional) libraries in the environment, by setting the values of the `HDF5_DIR`, `LZO_DIR`, `BZIP2_DIR` or `BLOSC_DIR` environment variables to the path to the particular resource. For example:

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

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

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

Development version (Unix)

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

1. sources have to be downloaded from the PyTables source repository hosted on GitHub. Git (see *[GIT]*) is used as VCS. The following command create a local copy of latest development version sources:

```
$ git clone https://github.com/PyTables/PyTables.git
```

2. sources in the git repository do not include pre-built documentation and pre-generated C code of Cython extension modules. To be able to generate them, both Cython (see *[CYTHON]*) and sphinx >= 1.0.7 (see *[SPHINX]*) are mandatory prerequisites.

2.1.2 PyTables package installation

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

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

```
$ python setup.py build
```

If the HDF5 installation is in a custom path, e.g. $HOME/hdf5-1.8.15pre7, one of the following commands can be used:

```
$ python setup.py build --hdf5=$HOME/hdf5-1.8.15pre7
```

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

Unix In the sh shell and its variants:

```
$ cd build/lib.linux-x86_64-3.3
$ env PYTHONPATH=. python tables/tests/test_all.py
```

or, if you prefer:

```
$ cd build/lib.linux-x86_64-3.3
$ env PYTHONPATH=. python -c "import tables; tables.test()"
```

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

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

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

Windows

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

```
> cd build\\lib.linux-x86_64-2.7
> set PYTHONPATH=.;%PYTHONPATH%
> python tables\\tests\\test_all.py
```

or:

```
> cd build\\lib.linux-x86_64-2.7
> set PYTHONPATH=.;%PYTHONPATH%
> python -c "import tables; tables.test()"
```

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

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

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

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

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

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

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

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

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

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

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

Unix

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

Windows

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

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

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

Again if one needs to point to libraries installed in custom paths, then specific setup.py options can be used:

```
$ python setup.py install --hdf5=/hdf5/custom/path
```

or:

```
$ env HDF5_DIR=/hdf5/custom/path 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 Installation with `pip`

Many users find it useful to use the **`pip`** program (or similar ones) to install python packages.

As explained in previous sections the user should in any case ensure that all dependencies listed in the *Prerequisites* section are correctly installed.

The simplest way to install PyTables using **pip** is the following:

```
$ pip install tables
```

The following example shows how to install the latest stable version of PyTables in the user folder when a older version of the package is already installed at system level:

```
$ pip install --user --upgrade tables
```

The *–user* option tells to the **pip** tool to install the package in the user folder (`$HOME/.local` on GNU/Linux and Unix systems), while the *–upgrade* option forces the installation of the latest version even if an older version of the package is already installed.

Additional options for the setup.py script can be specified using them *–install-option*:

```
$ pip install --install-option='--hdf5=/custom/path/to/hdf5' tables
```

or:

```
$ env HDF5_DIR=/custom/path/to/hdf5 pip install tables
```

The **pip** tool can also be used to install packages from a source tar-ball:

```
$ pip install tables-3.0.0.tar.gz
```

To install the development version of PyTables from the *develop* branch of the main **git** *[GIT]* repository the command is the following:

```
$ pip install git+https://github.com/PyTables/PyTables.git@develop#egg=tables
```

A similar command can be used to install a specific tagged fersion:

```
$ pip install git+https://github.com/PyTables/PyTables.git@v.2.4.0#egg=tables
```

Finally, PyTables developers provide a `requirements.txt` file that can be used by **pip** to install the PyTables dependencies:

```
$ wget https://raw.github.com/PyTables/PyTables/develop/requirements.txt
$ pip install -r requirements.txt
```

Of course the `requirements.txt` file can be used to install only python packages. Other dependencies like the HDF5 library of compression libraries have to be installed by the user.

Note: Recent versions of Debian and Ubuntu the HDF5 library is installed in with a very peculiar layout that allows to have both the serial and MPI versions installed at the same time.

PyTables >= 3.2 natively supports the new layout via *pkg-config* (that is expected to be installed on the system at build time).

If *pkg-config* is not available or PyTables is older that verison 3.2, then the following command can be used:

```
$ env CPPFLAGS=-I/usr/include/hdf5/serial \
LDFLAGS=-L/usr/lib/x86_64-linux-gnu/hdf5/serial python3 setup.py install
```

or:

```
$ env CPPFLAGS=-I/usr/include/hdf5/serial \
LDFLAGS=-L/usr/lib/x86_64-linux-gnu/hdf5/serial pip install tables
```

2.3 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.

2.3.1 Windows prerequisites

First, make sure that you have Python 2.6, NumPy 1.4.1 and Numexpr 2.0 or higher installed (PyTables binaries have been built using NumPy 1.7 and Numexpr 2.1). The binaries already include DLLs for HDF5 (1.8.4, 1.8.9), zlib1 (1.2.3), szlib (2.0, uncompression support only) and bzip2 (1.0.5) for Windows (2.8.0). The LZO DLL can't be included because of license issues (but read below for directives to install it if you want so).

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

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

2.3.2 PyTables package installation

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

Binary packahes can be found e.g. at the Unofficial Windows Binaries for Python Extension Packages page.

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

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

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

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

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

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

CHAPTER

THREE

TUTORIALS

> Seràs la clau que obre tots els panys, seràs la llum, la llum il.limitada, seràs confí on l'aurora comença, seràs forment, escala il.luminada!
>
> —Lyrics: Vicent Andrés i Estellés. Music: Ovidi Montllor, Toti Soler, M'aclame a tu

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

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

3.1 Getting started

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

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

3.1.1 Importing tables objects

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

```
>>> import tables
```

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

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

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

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

3.1.2 Declaring a Column Descriptor

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

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

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

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

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

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

3.1.3 Creating a PyTables file from scratch

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

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

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

3.1.4 Creating a new group

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

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

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

3.1.5 Creating a new table

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

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

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

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

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

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

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

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

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

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

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

Below is an example of how to write rows:

```
>>> for i in xrange(10):
...     particle['name']  = 'Particle: %6d' % (i)
...     particle['TDCcount'] = i % 256
...     particle['ADCcount'] = (i * 256) % (1 << 16)
...     particle['grid_i'] = i
...     particle['grid_j'] = 10 - i
...     particle['pressure'] = float(i*i)
...     particle['energy'] = float(particle['pressure'] ** 4)
...     particle['idnumber'] = i * (2 ** 34)
...     # Insert a new particle record
...     particle.append()
>>>
```

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

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

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

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

3.1.6 Reading (and selecting) data in a table

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

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

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

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

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

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

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

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

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

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

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

Note: A special care should be taken when the query condition includes string literals. Indeed Python 2 string literals are string of bytes while Python 3 strings are unicode objects.

With reference to the above definition of `Particle` it has to be noted that the type of the "name" column do not change depending on the Python version used (of course). It always corresponds to strings of bytes.

Any condition involving the "name" column should be written using the appropriate type for string literals in order to avoid `TypeErrors`.

Suppose one wants to get rows corresponding to specific particle names.

The code below will work fine in Python 2 but will fail with a `TypeError` in Python 3:

```
>>> condition = '(name == "Particle:      5") | (name == "Particle:      7")'
>>> for record in table.where(condition):  # TypeError in Python3
...     # do something with "record"
```

The reason is that in Python 3 "condition" implies a comparison between a string of bytes ("name" column contents) and an unicode literals.

The correct way to write the condition is:

```
>>> condition = '(name == b"Particle:      5") | (name == b"Particle:      7")'
```

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

3.1.7 Creating new array objects

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

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

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

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

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

As you can see, `File.create_array()` accepts *names* (which is a regular Python list) as an *object* parameter. Actually, it accepts a variety of different regular objects (see `create_array()`) 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 create_array method returns an Array instance that is not assigned to any variable. Don’t worry, this is intentional to show the kind of object we have created by displaying its representation. The Array objects have been attached to the object tree and saved to disk, as you can see if you print the complete object tree:

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

3.1.8 Closing the file and looking at its content

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

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

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

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

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

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

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

Also, in *Figure 1*, you can admire how the tutorial1.h5 looks like using the ViTables graphical interface.

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 = open_file("tutorial1.h5", "a")
```

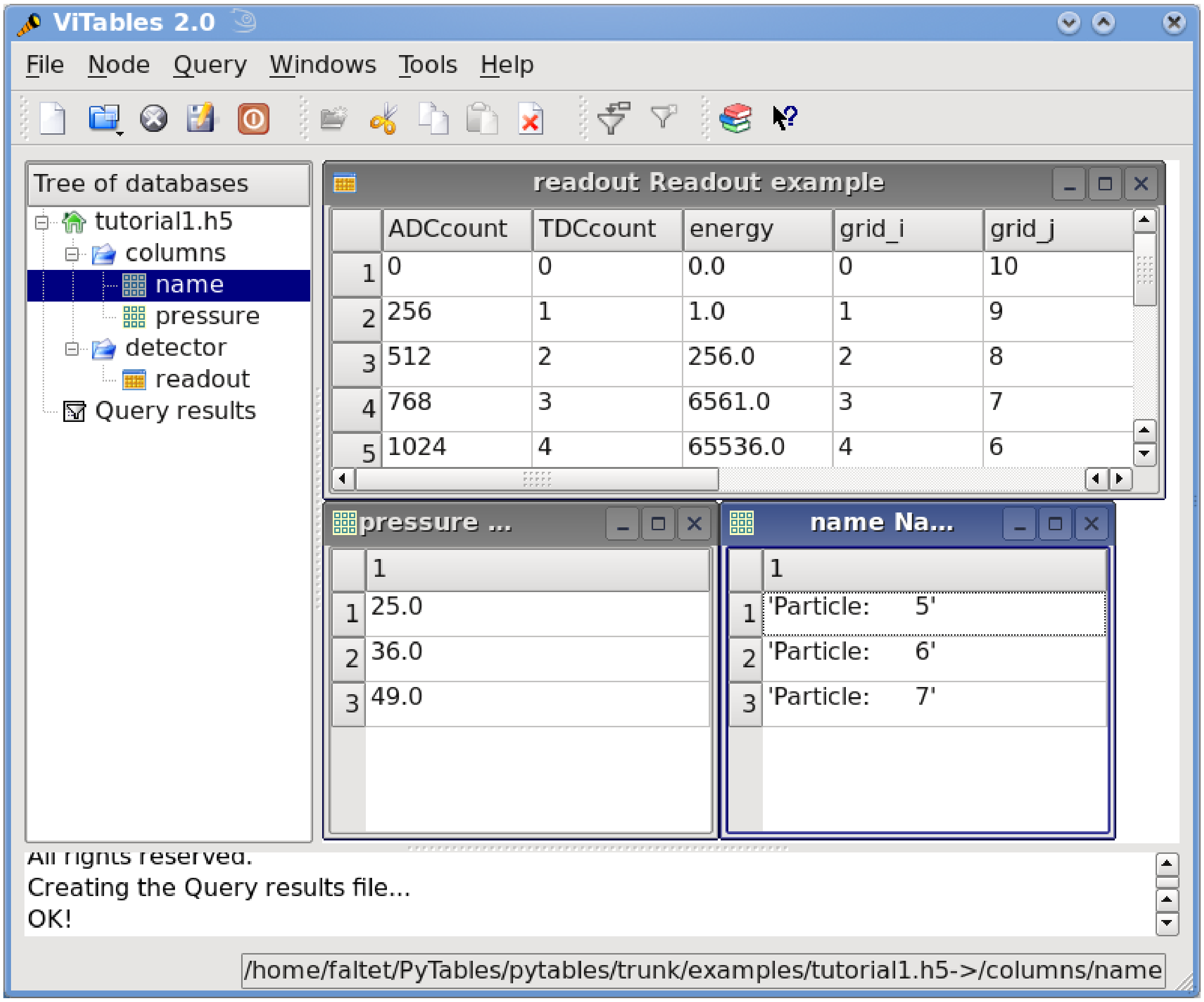
This time, we have opened the file in "a"ppend mode. We use this mode to add more information to the file.

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

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

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

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



	ADCcount	TDCcount	energy	grid_i	grid_j
1	0	0	0.0	0	10
2	256	1	1.0	1	9
3	512	2	256.0	2	8
4	768	3	6561.0	3	7
5	1024	4	65536.0	4	6

	1
1	25.0
2	36.0
3	49.0

	1
1	'Particle: 5'
2	'Particle: 6'
3	'Particle: 7'

```
All rights reserved.
Creating the Query results file...
OK!
```

```
/home/faltet/PyTables/pytables/trunk/examples/tutorial1.h5->/columns/name
```

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

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

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

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

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

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

`File.list_nodes()` returns a list containing all the nodes hanging off a specific Group. If the *classname* keyword is specified, the method will filter out all instances which are not descendants of the class. We have asked for only Array instances. There exist also an iterator counterpart called `File.iter_nodes()` 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 `File.walk_nodes()` special method of the File object. For example:

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

This is a nice shortcut when working interactively.

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

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

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

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

3.2.2 Setting and getting user attributes

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

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

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

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

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

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

Retrieving the attributes is equally simple:

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

You can probably guess how to delete attributes:

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

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

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

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

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

You can also rename attributes:

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

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

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

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

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

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

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

```
$ h5ls -vr tutorial1.h5/detector/readout
Opened "tutorial1.h5" with sec2 driver.
/detector/readout        Dataset {10/Inf}
    Attribute: CLASS     scalar
        Type:      6-byte null-terminated ASCII string
        Data:  "TABLE"
    Attribute: VERSION   scalar
        Type:      4-byte null-terminated ASCII string
        Data:  "2.6"
    Attribute: TITLE     scalar
        Type:      16-byte null-terminated ASCII string
        Data:  "Readout example"
    Attribute: NROWS     scalar
```

```
        Type:      native long long
        Data:  10
    Attribute: FIELD_0_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "ADCcount"
    Attribute: FIELD_1_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "TDCcount"
    Attribute: FIELD_2_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "energy"
    Attribute: FIELD_3_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "grid_i"
    Attribute: FIELD_4_NAME scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "grid_j"
    Attribute: FIELD_5_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "idnumber"
    Attribute: FIELD_6_NAME scalar
        Type:      5-byte null-terminated ASCII string
        Data:  "name"
    Attribute: FIELD_7_NAME scalar
        Type:      9-byte null-terminated ASCII string
        Data:  "pressure"
    Attribute: FLAVOR    scalar
        Type:      5-byte null-terminated ASCII string
        Data:  "numpy"
    Attribute: tempScale scalar
        Type:      7-byte null-terminated ASCII string
        Data:  "Celsius"
    Attribute: temperature scalar
        Type:      native double
        Data:  18.4
    Location:  0:1:0:1952
    Links:     1
    Modified:  2006-12-11 10:35:13 CET
    Chunks:    {85} 3995 bytes
    Storage:   470 logical bytes, 3995 allocated bytes, 11.76% utilization
    Type:      struct {
                    "ADCcount"         +0    native unsigned short
                    "TDCcount"         +2    native unsigned char
                    "energy"           +3    native double
                    "grid_i"           +11   native int
                    "grid_j"           +15   native int
                    "idnumber"         +19   native long long
                    "name"             +27   16-byte null-terminated ASCII string
                    "pressure"         +43   native float
                } 47 bytes
```

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

3.2.3 Getting object metadata

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

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

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

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

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

Try getting help with other object docs by yourself:

```
>>> help(h5file)
>>> help(table.remove_rows)
```

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

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

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

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

3.2.4 Reading data from Array objects

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

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

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

3.3 Commiting data to tables and arrays

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

3.3.1 Appending data to an existing table

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

```
>>> table = h5file.root.detector.readout
>>> particle = table.row
>>> for i in xrange(10, 15):
...     particle['name']  = 'Particle: %6d' % (i)
...     particle['TDCcount'] = i % 256
...     particle['ADCcount'] = (i * 256) % (1 << 16)
...     particle['grid_i'] = i
...     particle['grid_j'] = 10 - i
...     particle['pressure'] = float(i*i)
...     particle['energy'] = float(particle['pressure'] ** 4)
...     particle['idnumber'] = i * (2 ** 34)
...     particle.append()
>>> table.flush()
```

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

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

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

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

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

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

```
>>> for r in table.iterrows():
...     print("%-16s | %11.1f | %11.4g | %6d | %6d | %8d \|" % \\
...         (r['name'], r['pressure'], r['energy'], r['grid_i'], r['grid_j'],
...         r['TDCcount']))
Particle:      0 |         0.0 |           0 |      0 |     10 |        0 |
Particle:      1 |         1.0 |           1 |      1 |      9 |        1 |
Particle:      2 |         4.0 |         256 |      2 |      8 |        2 |
Particle:      3 |         9.0 |        6561 |      3 |      7 |        3 |
Particle:      4 |        16.0 |   6.554e+04 |      4 |      6 |        4 |
Particle:      5 |        25.0 |   3.906e+05 |      5 |      5 |        5 |
Particle:      6 |        36.0 |    1.68e+06 |      6 |      4 |        6 |
Particle:      7 |        49.0 |   5.765e+06 |      7 |      3 |        7 |
Particle:      8 |        64.0 |   1.678e+07 |      8 |      2 |        8 |
Particle:      9 |        81.0 |   4.305e+07 |      9 |      1 |        9 |
Particle:     10 |       100.0 |       1e+08 |     10 |      0 |       10 |
Particle:     11 |       121.0 |   2.144e+08 |     11 |     -1 |       11 |
Particle:     12 |       144.0 |     4.3e+08 |     12 |     -2 |       12 |
Particle:     13 |       169.0 |   8.157e+08 |     13 |     -3 |       13 |
Particle:     14 |       196.0 |   1.476e+09 |     14 |     -4 |       14 |
```

3.3.2 Modifying data in tables

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

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

We can modify complete ranges of columns as well:

```
>>> table.cols.TDCcount[2:5] = [2,3,4]
>>> print("After modifying slice [2:5] of TDCcount-->", table[0:5])
After modifying slice [2:5] of TDCcount-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (256, 1, 1.0, 1, 9, 17179869184L, 'Particle:      1', 1.0)
 (512, 2, 256.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)
 (768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)
 (1024, 4, 65536.0, 4, 6, 68719476736L, 'Particle:      4', 16.0)]
>>> table.cols.energy[1:9:3] = [2,3,4]
>>> print("After modifying slice [1:9:3] of energy-->", table[0:9])
```

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

Check that the values have been correctly modified!

Hint: remember that column TDCcount is the second one, and that energy is the third. Look for more info on modifying columns in *Column.__setitem__()*.

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

```
>>> table.modify_rows(start=1, step=3,
...                 rows=[(1, 2, 3.0, 4, 5, 6L, 'Particle:   None', 8.0),
...                       (2, 4, 6.0, 8, 10, 12L, 'Particle: None*2', 16.0)])
2
>>> print("After modifying the complete third row-->", table[0:5])
After modifying the complete third row-->
[(0, 1, 2.0, 0, 10, 0L, 'Particle:      0', 0.0)
 (1, 2, 3.0, 4, 5, 6L, 'Particle:   None', 8.0)
 (512, 2, 256.0, 2, 8, 34359738368L, 'Particle:      2', 4.0)
 (768, 3, 6561.0, 3, 7, 51539607552L, 'Particle:      3', 9.0)
 (2, 4, 6.0, 8, 10, 12L, 'Particle: None*2', 16.0)]
```

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

Apart of *Table.modify_rows()*, there exists another method, called *Table.modify_column()* to modify specific columns as well.

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

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

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

3.3.3 Modifying data in arrays

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

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

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

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

Similarly, with an array of strings:

```
>>> nameObject = h5file.root.columns.name
>>> print("Before modif-->", nameObject[:])
Before modif--> ['Particle:      5', 'Particle:      6', 'Particle:      7']
>>> nameObject[0] = 'Particle:   None'
>>> print("First modif-->", nameObject[:])
First modif--> ['Particle:   None', 'Particle:      6', 'Particle:      7']
>>> nameObject[1:3] = ['Particle:      0', 'Particle:      1']
>>> print("Second modif-->", nameObject[:])
Second modif--> ['Particle:   None', 'Particle:      0', 'Particle:      1']
>>> nameObject[::2] = ['Particle:     -3', 'Particle:     -5']
>>> print("Third modif-->", nameObject[:])
Third modif--> ['Particle:     -3', 'Particle:      0', 'Particle:     -5']
```

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

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

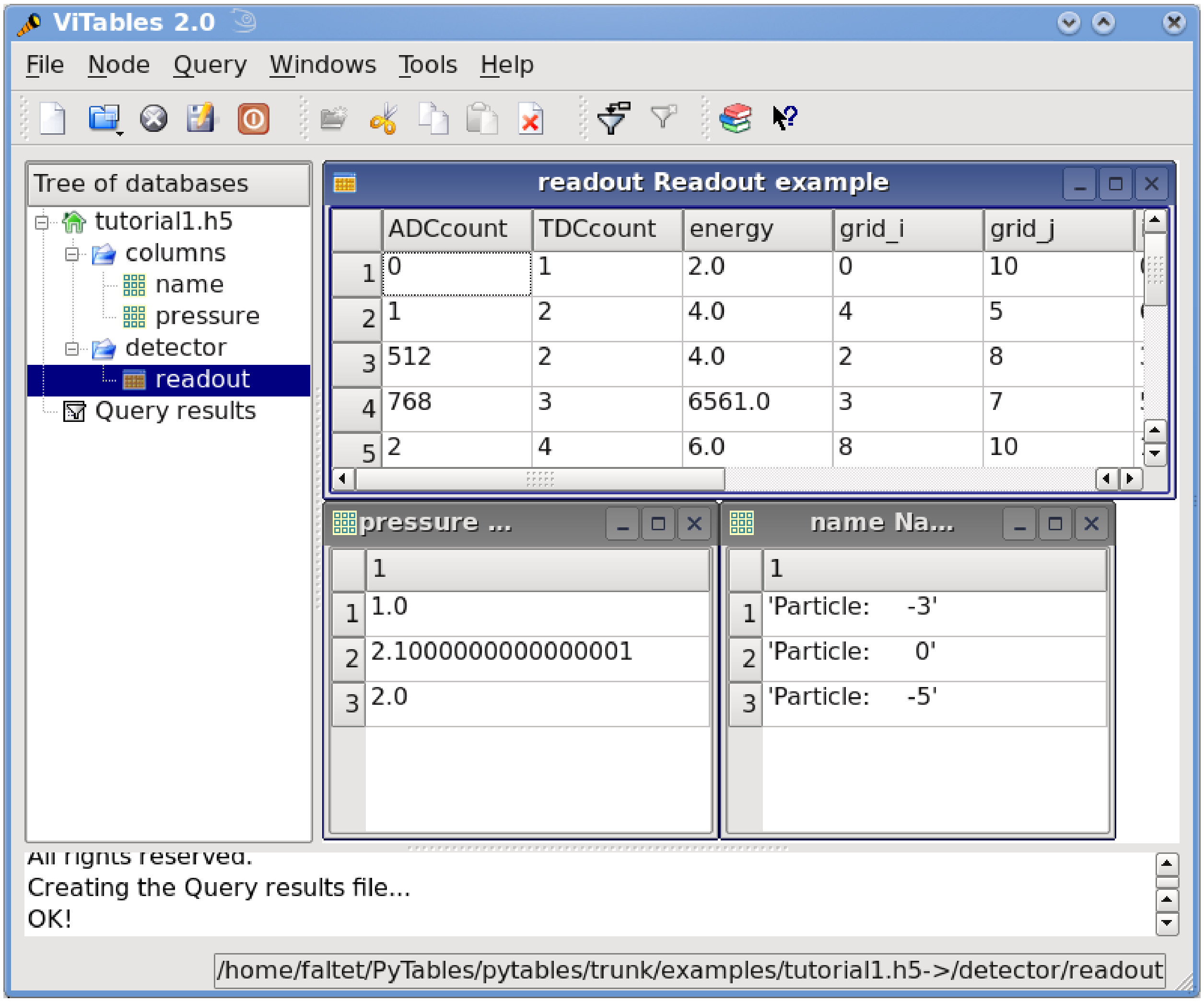
```
>>> table.remove_rows(5,10)
5
```

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

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

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

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



	ADCcount	TDCcount	energy	grid_i	grid_j
1	0	1	2.0	0	10
2	1	2	4.0	4	5
3	512	2	4.0	2	8
4	768	3	6561.0	3	7
5	2	4	6.0	8	10

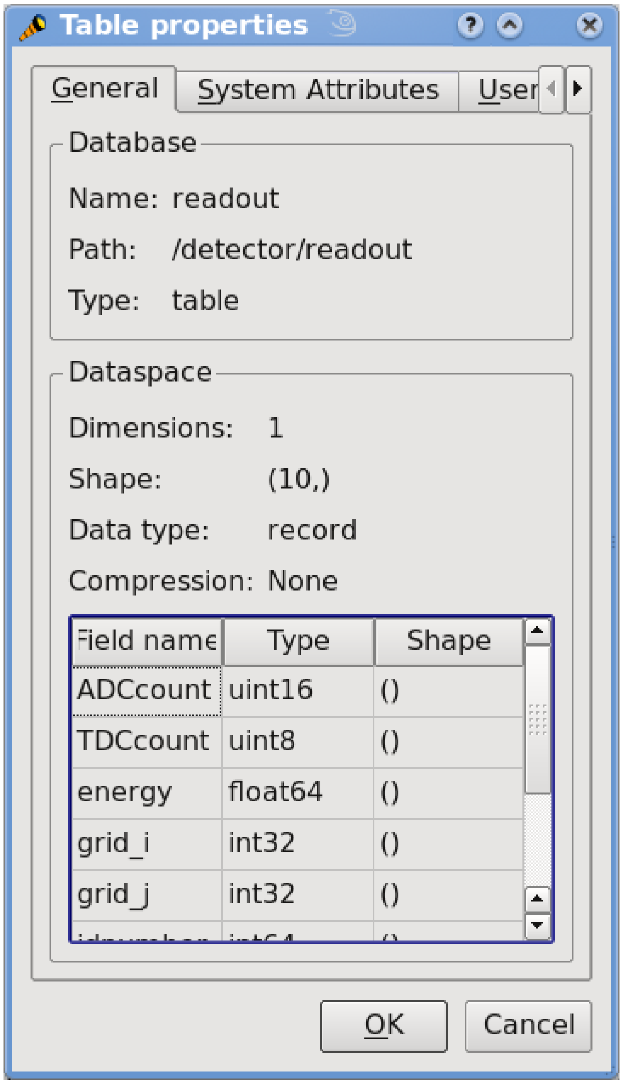
	1
1	1.0
2	2.1000000000000001
3	2.0

	1
1	'Particle: -3'
2	'Particle: 0'
3	'Particle: -5'

```
All rights reserved.
Creating the Query results file...
OK!
```

/home/faltet/PyTables/pytables/trunk/examples/tutorial1.h5->/detector/readout

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



Database

Name: readout

Path: /detector/readout

Type: table

Dataspace

Dimensions: 1

Shape: (10,)

Data type: record

Compression: None

Field name	Type	Shape
ADCcount	uint16	()
TDCcount	uint8	()
energy	float64	()
grid_i	int32	()
grid_j	int32	()

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

3.4 Multidimensional table cells and automatic sanity checks

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

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

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

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

```
from tables import *
from numpy import *

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

# Native NumPy dtype instances are also accepted
Event = dtype([
    ("name"     , "S16"),
    ("TDCcount" , uint8),
    ("ADCcount" , uint16),
    ("xcoord"   , float32),
    ("ycoord"   , float32)
    ])

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

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

# Get the HDF5 root group
root = fileh.root

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

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

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

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

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

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

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

        # This injects the Record values
        particle.append()

    # Flush the table buffers
    table.flush()

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

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

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

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

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

        # This injects the Record values
        event.append()

    # Flush the buffers
    table.flush()

# Read the records from table "/Events/TEvent3" and select some
```

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

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

3.4.1 Shape checking

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

```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 60, in <module>
    particle['pressure'] = array(i*arange(2*3)).reshape((2,4))  # Incorrect
ValueError: total size of new array must be unchanged
Closing remaining open files: tutorial2.h5... done
```

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

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

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

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

3.4.2 Field name checking

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

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

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

3.4.3 Data type checking

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

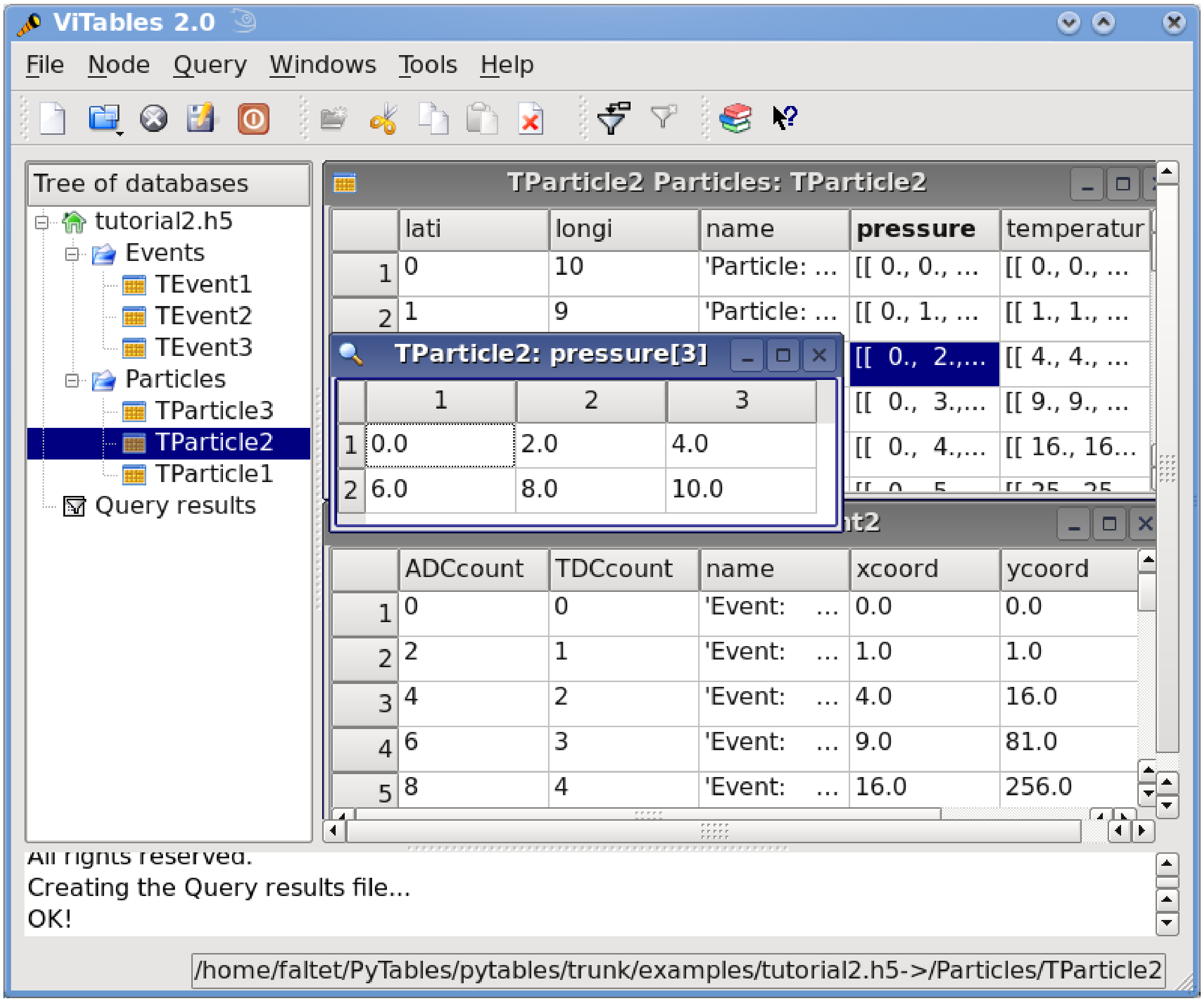
```
$ python tutorial2.py
Traceback (most recent call last):
  File "tutorial2.py", line 75, in ?
    event['ADCcount'] = "sss"          # Wrong type
  File "tableextension.pyx", line 1111, in tableextension.Row.__setitem__
TypeError: invalid type (<type 'str'>) for column ``ADCcount``
```

And, if we change the affected line to read:

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

we will see that the script ends well.

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



	lati	longi	name	pressure	temperatur
1	0	10	'Particle: ...	[[0., 0., ...	[[0., 0., ...
2	1	9	'Particle: ...	[[0., 1., ...	[[1., 1., ...
				[[0., 2.,...	[[4., 4., ...
				[[0., 3.,...	[[9., 9., ...
				[[0., 4.,...	[[16., 16...

	1	2	3
1	0.0	2.0	4.0
2	6.0	8.0	10.0

	ADCcount	TDCcount	name	xcoord	ycoord
1	0	0	'Event: ...	0.0	0.0
2	2	1	'Event: ...	1.0	1.0
3	4	2	'Event: ...	4.0	16.0
4	6	3	'Event: ...	9.0	81.0
5	8	4	'Event: ...	16.0	256.0

```
All rights reserved.
Creating the Query results file...
OK!
```

/home/faltet/PyTables/pytables/trunk/examples/tutorial2.h5->/Particles/TParticle2

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

3.5 Using links for more convenient access to nodes

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

Hard links let the user create additional paths to access another node in the same file, and once created, they are indistinguishable from the referred node object, except that they have different paths in the object tree. For example, if the referred node is, say, a Table object, then the new hard link will become a Table object itself. From this point on, you will be able to access the same Table object from two different paths: the original one and the new hard link path. If you delete one path to the table, you will be able to reach it via the other path.

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

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

3.5.1 Interactive example

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

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

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

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

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

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

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

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

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

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

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

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

Okay, we see how the first link /gl/la1 points to the array /g1/a1. Notice how the link prints as a SoftLink, and how the referred node is stored in the target instance attribute. The second link (/gt/lt) pointing to /g1/g2/t1 also has been created successfully, but by better inspecting the string representation of it, we see that is labeled as '(dangling)'. Why is this? Well, you should remember that we recently removed the /g1/g2/t1 path to access table t1. When printing it, the object knows that it points to *nowhere* and reports this. This is a nice way to quickly know whether a soft link points to an exiting node or not.

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

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

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

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

```
>>> plt = lt()
>>> print("dereferred lt node: ``%s``" % plt)
dereferred lt node: ``/g1/g2/t1 (Table(0,)) ``
>>> pla1 = la1()
>>> print("dereferred la1 node: ``%s``" % pla1)
dereferred la1 node: ``/g1/a1 (CArray(10000,)) ``
```

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

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

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

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

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

```
>>> la1.remove()
>>> la1 = f1.create_external_link(gl, 'la1', 'links2.h5:/a1')
>>> print("``%s`` is an external link to: ``%s``" % (la1, la1.target))
``/gl/la1 (ExternalLink) -> links2.h5:/a1`` is an external link to: ``links2.h5:/a1``
```

Let's try dereferring it:

```
>>> new_a1 = la1()  # dereferrencing la1 returns a1 in links2.h5
>>> print("dereferred la1 node:  ``%s``" % new_a1)
dereferred la1 node:  ``/a1 (CArray(10000,)) ``
```

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

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

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

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

```
>>> f1.close()
>>> exit()
$ ptdump links1.h5
/ (RootGroup) ''
/g1 (Group) ''
/g1/a1 (CArray(10000,)) ''
/gl (Group) ''
/gl/ht (Table(0,)) ''
/gl/la1 (ExternalLink) -> links2.h5:/a1
/gl/lt (SoftLink) -> /g1/g2/t1
/g1/g2 (Group) ''
/g1/g2/t1 (Table(0,)) ''
```

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

3.6 Exercising the Undo/Redo feature

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

You can undo/redo all the operations that are related to object tree management, like creating, deleting, moving or renaming nodes (or complete sub-hierarchies) inside a given object tree. You can also undo/redo operations (i.e. creation, deletion or modification) of persistent node attributes. However, when actions include *internal* modifications of datasets (that includes Table.append, Table.modify_rows or Table.remove_rows 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 [3].

3.6.1 A basic example

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

First, let's create a file:

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

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

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

```
>>> fileh.enable_undo()
```

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.create_array('/', '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 `File.mark()` for more info on mark()). Now, we are going to create another array:

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

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

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

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

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

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

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

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

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

```
>>> print(fileh.get_current_mark())
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.enable_undo(). Fine, but you are missing your too-young-to-die arrays. What can we do about that? `File.redo()` to the rescue:

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

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

```
>>> fileh.root.anarray.read()
[3, 4]
>>> fileh.root.anarray.title
'An array'
```

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

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

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

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

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

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

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

```
>>> fileh.disable_undo()
```

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

3.6.2 A more complete example

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

Let's introduce the first part of the code:

```
import tables

# Create an HDF5 file
```

```
fileh = tables.open_file('tutorial3-2.h5', 'w', title='Undo/Redo demo 2')

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

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

# Create a 'first' mark
fileh.mark('first')
fileh.create_array('/agroup', 'otherarray2', [4,5], 'Another array 2')
fileh.create_group('/agroup', 'agroup2', 'Group 2')

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

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

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

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

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

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

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

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

There are also a couple of implicit marks for going to the beginning or the end of the saved states: 0 and -1. Going to mark #0 means go to the beginning of the saved actions, that is, when method fileh.enable_undo() 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.disable_undo()
# Close the file
fileh.close()
```

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

3.7 Using enumerated types

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

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

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

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

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

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

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

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

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

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

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

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

3.7.1 Enumerated columns

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

```
>>> h5f = tables.open_file('enum.h5', 'w')
>>> class BallExt(tables.IsDescription):
...     ballTime = tables.Time32Col()
...     ballColor = tables.EnumCol(colors, 'black', base='uint8')
>>> tbl = h5f.create_table('/', '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 [4].

Let us use some random values to fill the table:

```
>>> import time
>>> import random
>>> now = time.time()
>>> row = tbl.row
>>> for i in range(10):
...     row['ballTime'] = now + i
...     row['ballColor'] = colors[random.choice(colorList)]  # notice this
...     row.append()
>>>
```

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

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

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

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

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

3.7.2 Enumerated arrays

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

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

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

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

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

Now let us feed some data to the array:

```
>>> wdays = earr.get_enum()
>>> 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 get_enum() (see *EArray methods*) to get it from the array (we could also have used dayRange.enum). Also note that we were able to append an invalid value (1234). Array methods do not check the validity of enumerated values.

Finally, we will print the contents of the array:

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

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

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

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

3.8 Dealing with nested structures in tables

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

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

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

```
from tables import *

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

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

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

    class info2(IsDescription):
        _v_pos = 1
        name = StringCol(10)
        value = Float64Col(pos=0)

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

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

3.8.1 Nested table creation

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

```
>>> fileh = open_file("nested-tut.h5", "w")
>>> table = fileh.create_table(fileh.root, 'table', NestedDescr)
```

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

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

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

3.8.2 Reading nested tables

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

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

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

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

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

Or, to create new tables:

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

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

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

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

3.8.3 Using Cols accessor

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

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

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

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

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

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

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

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

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

3.8.4 Accessing meta-information of nested tables

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

Let's see how it looks like:

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

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

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

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

There are other variables that can be interesting for you:

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

_v_nested_names 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_nested_descr, that can be useful to create nested structured arrays that imitate the structure of the table (or a subtable thereof):

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

You can see a simple example on how to create an array with NumPy.

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

```
>>> for coldescr in table.description._f_walk():
...     print("column-->",coldescr)
column--> Description([('info2', [('info3', [('x', '()f8'), ('y', '()u1')]),
                       ('name', '()S10'), ('value', '()f8')]),
                       ('info1', [('name', '()S10'), ('value', '()f8')]),
                       ('color', '()u4')])
column--> EnumCol(enum=Enum({'blue': 2, 'green': 1, 'red': 0}), dflt='red',
                            base=UInt32Atom(shape=(), dflt=0), shape=(), pos=2)
column--> Description([('info3', [('x', '()f8'), ('y', '()u1')]), ('name', '()S10'),
                       ('value', '()f8')])
column--> StringCol(itemsize=10, shape=(), dflt='', pos=1)
column--> Float64Col(shape=(), dflt=0.0, pos=2)
column--> Description([('name', '()S10'), ('value', '()f8')])
column--> StringCol(itemsize=10, shape=(), dflt='', pos=0)
column--> Float64Col(shape=(), dflt=0.0, pos=1)
column--> Description([('x', '()f8'), ('y', '()u1')])
column--> Float64Col(shape=(), dflt=1.0, pos=0)
column--> UInt8Col(shape=(), dflt=1, pos=1)
```

See the *The Description class* for the complete listing of attributes and methods of Description.

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

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

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

```
$ ptdump -d nested-tut.h5
/ (RootGroup) ''
/table (Table(13,)) ''
  Data dump:
[0] (((1.0, 0), 'name2-0', 0.0), ('name1-0', 0.0), 0L)
[1] (((1.0, 1), 'name2-1', 0.0), ('name1-1', 0.0), 1L)
[2] (((1.0, 2), 'name2-2', 0.0), ('name1-2', 0.0), 2L)
```

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

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

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

3.9 Other examples in PyTables distribution

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

CHAPTER

FOUR

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 open_file 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 is_pytables_file() and is_hdf5_file(), respectively. There exists also a function called which_lib_version() that informs about the versions of the underlying C libraries (for example, HDF5 or Zlib) and another called print_versions() that prints all the versions of the software that PyTables relies on. Finally, test() lets you run the complete test suite from a Python console interactively.

4.1 Top-level variables and functions

4.1.1 Global variables

`tables.`**`__version__`** **= '3.2.0rc3'**

The PyTables version number.

`tables.`**`hdf5_version`** **= '1.8.12'**

The underlying HDF5 library version number.

New in version 3.0.

4.1.2 Global functions

`tables.`**`copy_file`**(*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.copy_file()`* method, which acts on an already opened file. kwargs takes keyword arguments used to customize the copying process. See the documentation of *`File.copy_file()`* for a description of those arguments.

`tables.`**`is_hdf5_file`**(*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.

`tables.`**`is_pytables_file`**(*filename*)

Determine whether a file is in the PyTables format.

When successful, it returns the format version string if the file is a PyTables file, None otherwise. If there were problems identifying the file, an HDF5ExtError is raised.

`tables.`**`open_file`**(*filename*, *mode='r'*, *title=''*, *root_uep='/'*, *filters=None*, ***kwargs*)

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

Parameters **filename** : str

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

mode : str

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

title : str

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

root_uep : str

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

Changed in version 3.0: The *rootUEP* parameter has been renamed into *root_uep*.

filters : Filters

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

Notes

In addition, it recognizes the (lowercase) names of parameters present in `tables/parameters.py` as additional keyword arguments. See *PyTables parameter files* for a detailed info on the supported parameters.

Note: If you need to deal with a large number of nodes in an efficient way, please see *Getting the most from the node LRU cache* for more info and advices about the integrated node cache engine.

tables.**set_blosc_max_threads**(*nthreads*)

Set the maximum number of threads that Blosc can use.

This actually overrides the *tables.parameters.MAX_BLOSC_THREADS* setting in tables.parameters, so the new value will be effective until this function is called again or a new file with a different *tables.parameters.MAX_BLOSC_THREADS* value is specified.

Returns the previous setting for maximum threads.

tables.**print_versions**()

Print all the versions of software that PyTables relies on.

tables.**restrict_flavors**(*keep=['python']*)

Disable all flavors except those in keep.

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

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

tables.**split_type**(*type*)

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

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

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

tables.**test**(*verbose=False*, *heavy=False*)

Run all the tests in the test suite.

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

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

Return 0 (os.EX_OK) if all tests pass, 1 in case of failure

tables.**which_lib_version**(*name*)

Get version information about a C library.

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

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

tables.**silence_hdf5_messages**(*silence=True*)

Silence (or re-enable) messages from the HDF5 C library.

The *silence* parameter can be used control the behaviour and reset the standard HDF5 logging.

New in version 2.4.

4.2 File manipulation class

4.2.1 The File Class

class `tables.`**`File`**(*filename*, *mode='r'*, *title=''*, *root_uep='/'*, *filters=None*, ***kwargs*)

The in-memory representation of a PyTables file.

An instance of this class is returned when a PyTables file is opened with the `tables.open_file()` 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 root_uep attribute. Other attributes are available.

File objects support an *Undo/Redo mechanism* which can be enabled with the `File.enable_undo()` 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 `File.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 `File.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 `File.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 `File.disable_undo()` method.

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

Parameters **filename** : str

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

mode : str

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

title : str

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

root_uep : str

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

Changed in version 3.0: The *rootUEP* parameter has been renamed into *root_uep*.

filters : Filters

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

Notes

In addition, it recognizes the (lowercase) names of parameters present in `tables/parameters.py` as additional keyword arguments. See *PyTables parameter files* for a detailed info on the supported parameters.

File attributes

`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.

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

`root_uep`
The UEP (user entry point) group name in the file (see the `open_file()` function).

Changed in version 3.0: The *rootUEP* attribute has been renamed into *root_uep*.

File properties

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

`File.`**`filters`**
Default filter properties for the root group (see *The Filters class*).

File.**open_count**

The number of times this file handle has been opened.

Changed in version 3.1: The mechanism for caching and sharing file handles has been removed in PyTables 3.1. Now this property should always be 1 (or 0 for closed files).

Deprecated since version 3.1.

File methods - file handling

File.**close**()

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

File.**copy_file**(*dstfilename*, *overwrite=False*, ***kwargs*)

Copy the contents of this file to dstfilename.

Parameters **dstfilename** : str

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.

overwrite : bool, optional

If true, the destination file will be overwritten if it already exists. In this case, the destination file must be closed, or errors will occur. Defaults to False.

kwargs :

Additional keyword arguments discussed below.

Notes

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.

In addition, it recognizes the names of parameters present in `tables/parameters.py` as additional keyword arguments. See *PyTables parameter files* for a detailed info on the supported parameters.

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

File.**flush**()

Flush all the alive leaves in the object tree.

File.**fileno**()

Return the underlying OS integer file descriptor.

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

File.**__enter__**()

Enter a context and return the same file.

File.**__exit__**(**exc_info*)

Exit a context and close the file.

File.**__str__**()

Return a short string representation of the object tree.

Examples

```
>>> f = tables.open_file('data/test.h5')
>>> print(f)
data/test.h5 (File) 'Table Benchmark'
Last modif.: 'Mon Sep 20 12:40:47 2004'
Object Tree:
/ (Group) 'Table Benchmark'
/tuple0 (Table(100,)) 'This is the table title'
/group0 (Group) ''
/group0/tuple1 (Table(100,)) 'This is the table title'
/group0/group1 (Group) ''
/group0/group1/tuple2 (Table(100,)) 'This is the table title'
/group0/group1/group2 (Group) ''
```

File.**__repr__**()

Return a detailed string representation of the object tree.

File.**get_file_image**()

Retrieves an in-memory image of an existing, open HDF5 file.

Note: this method requires HDF5 >= 1.8.9.

New in version 3.0.

File.**get_filesize**()

Returns the size of an HDF5 file.

The returned size is that of the entire file, as opposed to only the HDF5 portion of the file. I.e., size includes the user block, if any, the HDF5 portion of the file, and any data that may have been appended beyond the data written through the HDF5 Library.

New in version 3.0.

File.**get_userblock_size**()

Retrieves the size of a user block.

New in version 3.0.

File methods - hierarchy manipulation

File.**copy_children**(*srcgroup*, *dstgroup*, *overwrite=False*, *recursive=False*, *createparents=False*, ***kwargs*)

Copy the children of a group into another group.

Parameters **srcgroup** : str

The group to copy from.

dstgroup : str

The destination group.

overwrite : bool, optional

If True, the destination group will be overwritten if it already exists. Defaults to False.

recursive : bool, optional

If True, all descendant nodes of srcgroup are recursively copied. Defaults to False.

createparents : bool, optional

If True, any necessary parents of dstgroup will be created. Defaults to False.

kwargs : dict

Additional keyword arguments can be used to customize the copying process. See the documentation of `Group._f_copy_children()` for a description of those arguments.

`File.`**`copy_node`**(*where*, *newparent=None*, *newname=None*, *name=None*, *overwrite=False*, *recursive=False*, *createparents=False*, ***kwargs*)

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

Parameters **where** : str

These arguments work as in `File.get_node()`, referencing the node to be acted upon.

newparent : str or Group

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

newname : str

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

name : str

These arguments work as in `File.get_node()`, referencing the node to be acted upon.

overwrite : bool, optional

If True, the destination group will be overwritten if it already exists. Defaults to False.

recursive : bool, optional

If True, all descendant nodes of srcgroup are recursively copied. Defaults to False.

createparents : bool, optional

If True, any necessary parents of dstgroup will be created. Defaults to False.

kwargs :

Additional keyword arguments can be used to customize the copying process. See the documentation of `Group._f_copy()` for a description of those arguments.

Returns **node** : Node

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

`File.`**`create_array`**(*where*, *name*, *obj=None*, *title='‘*, *byteorder=None*, *createparents=False*, *atom=None*, *shape=None*)

Create a new array.

Parameters **where** : str or Group

The parent group from which the new array will hang. It can be a path string (for example ‘/level1/leaf5’), or a Group instance (see *The Group class*).

name : str

The name of the new array

obj : python object

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

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

Changed in version 3.0: The *Object parameter has been renamed into *obj.**

title : str

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

byteorder : str

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

createparents : bool, optional

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

atom : Atom

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

New in version 3.0.

shape : tuple of ints

The shape of the stored array.

New in version 3.0.

See also:

`Array` for more information on arrays

`create_table` for more information on the rest of parameters

`File.`**`create_carray`**(*where*, *name*, *atom=None*, *shape=None*, *title=''*, *filters=None*, *chunkshape=None*, *byteorder=None*, *createparents=False*, *obj=None*)

Create a new chunked array.

Parameters **where** : str or Group

The parent group from which the new array will hang. It can be a path string (for example '/level1/leaf5'), or a Group instance (see *The Group class*).

name : str

The name of the new array

atom : Atom

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

Changed in version 3.0: The *atom* parameter can be None (default) if *obj* is provided.

shape : tuple

The shape of the new array.

Changed in version 3.0: The *shape* parameter can be None (default) if *obj* is provided.

title : str, optional

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

filters : Filters, optional

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

chunkshape : tuple or number or None, optional

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

byteorder : str, optional

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

createparents : bool, optional

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

obj : python object

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

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

The *obj* parameter is optional and it can be provided in alternative to the *atom* and *shape* parameters. If both *obj* and *atom* and/or *shape* are provided they must be consistent with each other.

New in version 3.0.

See also:

`CArray` for more information on chunked arrays

`File.`**`create_earray`**(*where*, *name*, *atom=None*, *shape=None*, *title=''*, *filters=None*, *expectedrows=1000*, *chunkshape=None*, *byteorder=None*, *createparents=False*, *obj=None*)

Create a new enlargeable array.

Parameters **where** : str or Group

The parent group from which the new array will hang. It can be a path string (for example '/level1/leaf5'), or a Group instance (see *The Group class*).

name : str

The name of the new array

atom : Atom

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

Changed in version 3.0: The *atom* parameter can be None (default) if *obj* is provided.

shape : tuple

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

Changed in version 3.0: The *shape* parameter can be None (default) if *obj* is provided.

title : str, optional

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

expectedrows : int, optional

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

chunkshape : tuple, numeric, or None, optional

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

byteorder : str, optional

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

createparents : bool, optional

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

obj : python object

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

The *obj* parameter is optional and it can be provided in alternative to the *atom* and *shape* parameters. If both *obj* and *atom* and/or *shape* are provided they must be consistent with each other.

New in version 3.0.

See also:

EArray for more information on enlargeable arrays

`File.`**`create_external_link`**`(`*where*, *name*, *target*, *createparents=False*`)`
Create an external link.

Create an external link to a *target* node with the given *name* in *where* location. *target* can be a node object in another file or a path string in the form 'file:/path/to/node'. If *createparents* is true, the intermediate groups required for reaching *where* are created (the default is not doing so).

The returned node is an `ExternalLink` instance.

`File.`**`create_group`**(*where*, *name*, *title=''*, *filters=None*, *createparents=False*)

Create a new group.

Parameters **where** : str or Group

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

name : str

The name of the new group.

title : str, optional

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

filters : Filters

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

createparents : bool

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

See also:

`Group` for more information on groups

`File.`**`create_hard_link`**(*where*, *name*, *target*, *createparents=False*)

Create a hard link.

Create a hard link to a *target* node with the given *name* in *where* location. *target* can be a node object or a path string. If *createparents* is true, the intermediate groups required for reaching *where* are created (the default is not doing so).

The returned node is a regular *Group* or *Leaf* instance.

`File.`**`create_soft_link`**(*where*, *name*, *target*, *createparents=False*)

Create a soft link (aka symbolic link) to a *target* node.

Create a soft link (aka symbolic link) to a *target* nodewith the given *name* in *where* location. *target* can be a node object or a path string. If *createparents* is true, the intermediate groups required for reaching *where* are created.

(the default is not doing so).

The returned node is a SoftLink instance. See the SoftLink class (in *The SoftLink class*) for more information on soft links.

`File.`**`create_table`**(*where*, *name*, *description=None*, *title=''*, *filters=None*, *expectedrows=10000*, *chunkshape=None*, *byteorder=None*, *createparents=False*, *obj=None*)

Create a new table with the given name in where location.

Parameters **where** : str or Group

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

name : str

The name of the new table.

description : Description

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

- *A user-defined class*: This should inherit from the IsDescription class (see *The IsDescription class*) where table fields are specified.
- *A dictionary*: For example, when you do not know beforehand which structure your table will have).
- *A Description instance*: You can use the description attribute of another table to create a new one with the same structure.
- *A NumPy dtype*: A completely general structured NumPy dtype.
- *A NumPy (structured) array instance*: The dtype of this structured array will be used as the description. Also, in case the array has actual data, it will be injected into the newly created table.

Changed in version 3.0: The *description* parameter can be None (default) if *obj* is provided. In that case the structure of the table is deduced by *obj*.

title : str

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

filters : Filters

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

expectedrows : int

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

chunkshape :

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

byteorder : str

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

createparents : bool

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

obj : python object

The recarray to be saved. Accepted types are NumPy record arrays, as well as native Python sequences convertible to numpy record arrays.

The *obj* parameter is optional and it can be provided in alternative to the *description* parameter. If both *obj* and *description* are provided they must be consistent with each other.

New in version 3.0.

See also:

`Table` for more information on tables

`File.create_vlarray(where, name, atom=None, title='', filters=None, expectedrows=None, chunkshape=None, byteorder=None, createparents=False, obj=None)`

Create a new variable-length array.

Parameters **where** : str or Group

The parent group from which the new array will hang. It can be a path string (for example '/level1/leaf5'), or a Group instance (see *The Group class*).

name : str

The name of the new array

atom : Atom

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

Changed in version 3.0: The *atom* parameter can be None (default) if *obj* is provided.

title : str, optional

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

filters : Filters

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

expectedrows : int, optional

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

New in version 3.0.

chunkshape : int or tuple of int, optional

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

byteorder : str, optional

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

createparents : bool, optional

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

obj : python object

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

The *obj* parameter is optional and it can be provided in alternative to the *atom* parameter. If both *obj* and *atom* and are provided they must be consistent with each other.

New in version 3.0.

See also:

VLArray for more informationon variable-length arrays

The *expectedsizeinMB* parameter has been replaced by *expectedrows*.

File.**move_node**(*where*, *newparent=None*, *newname=None*, *name=None*, *overwrite=False*, *createparents=False*)

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

Parameters **where, name** : path

These arguments work as in `File.get_node()`, referencing the node to be acted upon.

newparent :

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

newname :

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

Notes

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

File.**remove_node**(*where*, *name=None*, *recursive=False*)

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

Parameters **where, name** :

These arguments work as in `File.get_node()`, referencing the node to be acted upon.

recursive : bool

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

File.**rename_node**(*where*, *newname*, *name=None*, *overwrite=False*)

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

Parameters **where, name** :

These arguments work as in `File.get_node()`, referencing the node to be acted upon.

newname : str

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

overwrite : bool

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

File methods - tree traversal

`File.`**`get_node`**`(`*where*, *name=None*, *classname=None*`)`

Get the node under where with the given name.

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

If a name is specified, this must be a string with the name of a node under where. In this case the where argument can only lead to a Group (see *The Group class*) 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.

`File.`**`is_visible_node`**`(`*path*`)`

Is the node under *path* visible?

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

`File.`**`iter_nodes`**`(`*where*, *classname=None*`)`

Iterate over children nodes hanging from where.

Parameters **where** :

This argument works as in `File.get_node()`, referencing the node to be acted upon.

classname :

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

Notes

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

`File.`**`list_nodes`**`(`*where*, *classname=None*`)`

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

This is a list-returning version of `File.iter_nodes()`.

`File.`**`walk_groups`**`(`*where='/'*`)`

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

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

The where argument can be a path string or a Group instance (see *The Group class*).

`File.`**`walk_nodes`**`(`*where='/'*, *classname=None*`)`

Recursively iterate over nodes hanging from where.

Parameters **where** : str or Group, optional

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

classname :

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

Notes

This version iterates over the leaves in the same group in order to avoid having a list referencing to them and thus, preventing the LRU cache to remove them after their use.

Examples

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

File.**__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.

File.**__iter__**()

Recursively iterate over the nodes in the tree.

This is equivalent to calling `File.walk_nodes()` with no arguments.

Examples

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

File methods - Undo/Redo support

File.**disable_undo**()

Disable the Undo/Redo mechanism.

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

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

File.**enable_undo**(*filters=Filters(complevel=1, complib='zlib', shuffle=True, fletcher32=False, least_significant_digit=None)*)

Enable the Undo/Redo mechanism.

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

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

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

`File.`**`get_current_mark`**`()`

Get the identifier of the current mark.

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

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

`File.`**`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.

`File.`**`is_undo_enabled`**`()`

Is the Undo/Redo mechanism enabled?

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

`File.`**`mark`**`(`*name=None*`)`

Mark the state of the database.

Creates a mark for the current state of the database. A unique (and immutable) identifier for the mark is returned. An optional name (a string) can be assigned to the mark. Both the identifier of a mark and its name can be used in `File.undo()` and `File.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.

`File.`**`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.

`File.`**`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.

File methods - attribute handling

`File.`**`copy_node_attrs`**`(`*where*, *dstnode*, *name=None*`)`

Copy PyTables attributes from one node to another.

Parameters **where, name** :

These arguments work as in *File.get_node()*, referencing the node to be acted upon.

dstnode :

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

`File.`**`del_node_attr`**`(`*where*, *attrname*, *name=None*`)`

Delete a PyTables attribute from the given node.

Parameters **where, name** :

These arguments work as in *File.get_node()*, referencing the node to be acted upon.

attrname :

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

`File.`**`get_node_attr`**`(`*where*, *attrname*, *name=None*`)`

Get a PyTables attribute from the given node.

Parameters **where, name** :

These arguments work as in *File.get_node()*, referencing the node to be acted upon.

attrname :

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

`File.`**`set_node_attr`**`(`*where*, *attrname*, *attrvalue*, *name=None*`)`

Set a PyTables attribute for the given node.

Parameters **where, name** :

These arguments work as in *File.get_node()*, referencing the node to be acted upon.

attrname :

The name of the attribute to set.

attrvalue :

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

Notes

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

4.3 Hierarchy definition classes

4.3.1 The Node class

class `tables.`**`Node`**(*parentnode*, *name*, *_log=True*)

Abstract base class for all PyTables nodes.

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

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

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

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

Sub-classes with no children (e.g. *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.

Node attributes

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

`_v_file`
The hosting File instance (see *The File Class*).

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

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

`_v_objectid`
A node identifier (may change from run to run).

Changed in version 3.0: The *_v_objectID* attribute has been renamed into *_v_object_id*.

Node instance variables - location dependent

`Node.`**`_v_parent`**

The parent `Group` instance

Node instance variables - location independent

`Node.`**`_v_attrs`**

The associated *AttributeSet* instance.

See also:

`tables.attributeset.AttributeSet` container for the HDF5 attributes

Node.**_v_isopen** = False

Whether this node is open or not.

Node instance variables - attribute shorthands

Node.**_v_title**

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

Node methods - hierarchy manipulation

Node.**_f_close**()

Close this node in the tree.

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

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

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

Copy this node and return the new node.

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

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

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

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

Node.**_f_isvisible**()

Is this node visible?

Node.**_f_move**(*newparent=None*, *newname=None*, *overwrite=False*, *createparents=False*)

Move or rename this node.

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

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

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

Node.**_f_remove**(*recursive=False*, *force=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.

If the node is a link to a Group object, and you are sure that you want to delete it, you can do this by setting the force flag to true.

Node.**_f_rename**(*newname*, *overwrite=False*)
Rename this node in place.

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

Node methods - attribute handling

Node.**_f_delattr**(*name*)
Delete a PyTables attribute from this node.

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

Node.**_f_getattr**(*name*)
Get a PyTables attribute from this node.

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

Node.**_f_setattr**(*name*, *value*)
Set a PyTables attribute for this node.

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

4.3.2 The Group class

class tables.**Group**(*parentnode*, *name*, *title=''*, *new=False*, *filters=None*, *_log=True*)
Basic PyTables grouping structure.

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

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

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

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

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

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*

name : str

The name of this node in its parent group.

title :

The title for this group

new :

If this group is new or has to be read from disk

filters : Filters

A Filters instance

Notes

The following documentation includes methods that are automatically called when a Group instance is accessed in a special way.

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

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

# Add a Table child called 'table' under 'group'.
h5file.create_table(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
```

Group attributes

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

`_v_children`
Dictionary with all nodes hanging from this group.

`_v_groups`
Dictionary with all groups hanging from this group.

`_v_hidden`
Dictionary with all hidden nodes hanging from this group.

_v_leaves
Dictionary with all leaves hanging from this group.

_v_links
Dictionary with all links hanging from this group.

_v_unknown
Dictionary with all unknown nodes hanging from this group.

Group properties

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

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

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

Group methods

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

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

Group.**_f_close**()
Close this group and all its descendents.

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

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

Group.**_f_copy**(*newparent=None*, *newname=None*, *overwrite=False*, *recursive=False*, *createparents=False*, ***kwargs*)
Copy this node and return the new one.

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

Parameters 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 : 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 *The Filters class*). The default is to copy the filter properties from the source node.

copyuserattrs :

You can prevent the user attributes from being copied by setting thisparameter 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', 'links' and 'bytes' having a numeric value. Their values willbe incremented to reflect the number of groups, leaves and bytes, respectively, that have been copied during the operation.

`Group.`**`_f_copy_children`**(*dstgroup*, *overwrite=False*, *recursive=False*, *createparents=False*, ***kwargs*)

Copy the children of this group into another group.

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

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

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

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

`Group.`**`_f_get_child`**(*childname*)

Get the child called childname of this group.

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

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

`Group.`**`_f_iter_nodes`**(*classname=None*)

Iterate over children nodes.

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

This is an iterator version of `Group._f_list_nodes()`.

`Group.`**`_f_list_nodes`**(*classname=None*)

Return a *list* with children nodes.

This is a list-returning version of `Group._f_iter_nodes()`.

`Group.`**`_f_walk_groups`**()

Recursively iterate over descendent groups (not leaves).

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

`Group.`**`_f_walknodes`**(*classname=None*)

Iterate over descendant nodes.

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

If *classname* is Group, it behaves like `Group._f_walk_groups()`, yielding only groups. If you don't want a recursive behavior, use `Group._f_iter_nodes()` instead.

Examples

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

Group special methods

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

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

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

# Add a Table child called 'table' under 'group'.
h5file.create_table(group, 'table', my_description)
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
```

`Group.`**`__contains__`**(*name*)

Is there a child with that *name*?

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

`Group.`**`__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.remove_node()` or `Node._f_remove()`. It does *neither* delete a PyTables node attribute; for that, use `File.del_node_attr()`, `Node._f_delattr()` or `Node._v_attrs`\`.

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

`Group.`**`__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.

`Group.`**`__iter__`**()

Iterate over the child nodes hanging directly from the group.

This iterator is *not* recursive.

Examples

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

Group.**__repr__**()

Return a detailed string representation of the group.

Examples

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

Group.**__setattr__**(*name*, *value*)

Set a Python attribute called name with the given value.

This method stores an *ordinary Python attribute* in the object. It does *not* store new children nodes under this group; for that, use the File.create*() methods (see the File class in *The File Class*). It does *neither* store a PyTables node attribute; for that, use `File.set_node_attr()`, :meth‘:Node._f_setattr‘ or `Node._v_attrs`.

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.get_node()`, `Group._f_get_child()` and children dictionaries in the group (if visible).

Group.**__str__**()

Return a short string representation of the group.

Examples

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

4.3.3 The Leaf class

class tables.**Leaf**(*parentnode*, *name*, *new=False*, *filters=None*, *byteorder=None*, *_log=True*)

Abstract base class for all PyTables leaves.

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

This definition includes all nodes which contain actual data (datasets handled by the Table - see *The Table class*, Array - see *The Array class*, CArray - see *The CArray class*, EArray - see *The EArray class*, and VLArray - see *The VLArray class* classes) and unsupported nodes (the UnImplemented class - *The UnImplemented class*) these classes do in fact inherit from Leaf.

Leaf attributes

These instance variables are provided in addition to those in Node (see *The Node class*):

byteorder
The byte ordering of the leaf data *on disk*. It will be either `little` or `big`.

dtype
The NumPy dtype that most closely matches this leaf type.

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

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

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

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

shape
The shape of data in the leaf.

Leaf properties

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

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

`Leaf.`**ndim**
The number of dimensions of the leaf data.

`Leaf.`**filters**
Filter properties for this leaf.

See also:

Filters

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

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

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

It can be any of ‘numpy’ or ‘python’.

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

`Leaf.`**size_in_memory**
The size of this leaf’s data in bytes when it is fully loaded into memory.

`Leaf.`**size_on_disk**
The size of this leaf’s data in bytes as it is stored on disk. If the data is compressed, this shows the compressed size. In the case of uncompressed, chunked data, this may be slightly larger than the amount of data, due to partially filled chunks.

Leaf instance variables - aliases

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

`Leaf.`**`attrs`**
The associated AttributeSet instance - see *The AttributeSet class* (This is an easier-to-write alias of `Node._v_attrs`.

`Leaf.`**`name`**
The name of this node in its parent group (This is an easier-to-write alias of `Node._v_name`).

`Leaf.`**`object_id`**
A node identifier, which may change from run to run. (This is an easier-to-write alias of `Node._v_objectid`).

Changed in version 3.0: The *objectID* property has been renamed into *object_id*.

`Leaf.`**`title`**
A description for this node (This is an easier-to-write alias of `Node._v_title`).

Leaf methods

`Leaf.`**`close`**(*flush=True*)
Close this node in the tree.

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

`Leaf.`**`copy`**(*newparent=None*, *newname=None*, *overwrite=False*, *createparents=False*, ***kwargs*)
Copy this node and return the new one.

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

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

Parameters **title** :

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

filters : 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 *The Filters class*). 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 : int

Specify the range of rows to be copied; the default is to copy all the rows.

stats :

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

chunkshape :

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

`Leaf.`**`flush`**`()`

Flush pending data to disk.

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

`Leaf.`**`isvisible`**`()`

Is this node visible?

This method has the behavior described in `Node._f_isvisible()`.

`Leaf.`**`move`**`(`*newparent=None*, *newname=None*, *overwrite=False*, *createparents=False*`)`

Move or rename this node.

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

`Leaf.`**`rename`**`(`*newname*`)`

Rename this node in place.

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

`Leaf.`**`remove`**`()`

Remove this node from the hierarchy.

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

`Leaf.`**`get_attr`**`(`*name*`)`

Get a PyTables attribute from this node.

This method has the behavior described in `Node._f_getattr()`.

`Leaf.`**`set_attr`**`(`*name*, *value*`)`

Set a PyTables attribute for this node.

This method has the behavior described in `Node._f_setattr()`.

`Leaf.`**`del_attr`**`(`*name*`)`

Delete a PyTables attribute from this node.

This method has the behavior described in `Node_f_delAttr()`.

`Leaf.`**`truncate`**`(`*size*`)`

Truncate the main dimension to be size rows.

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

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

`Leaf.`**`__len__`**`()`

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

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

`Leaf._f_close`(*flush=True*)

Close this node in the tree.

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

4.4 Structured storage classes

4.4.1 The Table class

class `tables.Table`(*parentnode*, *name*, *description=None*, *title=''*, *filters=None*, *expectedrows=None*, *chunkshape=None*, *byteorder=None*, *_log=True*)

This class represents heterogeneous datasets in an HDF5 file.

Tables are leaves (see the Leaf class in *The Leaf class*) whose data consists of a unidimensional sequence of *rows*, where each row contains one or more *fields*. Fields have an associated unique *name* and *position*, with the first field having position 0. All rows have the same fields, which are arranged in *columns*.

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

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

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

PyTables supports *in-kernel* searches working simultaneously on several columns using complex conditions. These are faster than selections using Python expressions. See the `Table.where()` method for more information on in-kernel searches.

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

When iterating a table, an object from the Row (see *The Row class*) class is used. This object allows to read and write data one row at a time, as well as to perform queries which are not supported by in-kernel syntax (at a much lower speed, of course).

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

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

description :

An IsDescription subclass or a dictionary where the keys are the field names, and the values the type definitions. In addition, a pure NumPy dtype is accepted. If None, the table metadata is read from disk, else, it's taken from previous parameters.

title :

Sets a TITLE attribute on the HDF5 table entity.

filters : Filters

An instance of the Filters class that provides information about the desired I/O filters to be applied during the life of this object.

expectedrows :

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

chunkshape :

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

byteorder :

The byteorder of the data *on-disk*, specified as 'little' or 'big'. If this is not specified, the byteorder is that of the platform, unless you passed a recarray as the *description*, in which case the recarray byteorder will be chosen.

Notes

The instance variables below are provided in addition to those in Leaf (see *The Leaf class*). Please note that there are several col* dictionaries to ease retrieving information about a column directly by its path name, avoiding the need to walk through Table.description or Table.cols.

Table attributes

`coldescrs`
Maps the name of a column to its Col description (see *The Col class and its descendants*).

`coldflts`
Maps the name of a column to its default value.

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

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

`colinstances`
Maps the name of a column to its Column (see *The Column class*) or Cols (see *The Cols class*) instance.

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

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

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

cols
A Cols instance that provides *natural naming* access to non-nested (Column, see *The Column class*) and nested (Cols, see *The Cols class*) columns.

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

description
A Description instance (see *The Description class*) reflecting the structure of the table.

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

indexed
Does this table have any indexed columns?

nrows
The current number of rows in the table.

Table properties

`Table.`**`autoindex`**
Automatically keep column indexes up to date?

Setting this value states whether existing indexes should be automatically updated after an append operation or recomputed after an index-invalidating operation (i.e. removal and modification of rows). The default is true.

This value gets into effect whenever a column is altered. If you don't have automatic indexing activated and you want to do an immediate update use *`Table.flush_rows_to_index()`*; for immediate reindexing of invalidated indexes, use *`Table.reindex_dirty()`*.

This value is persistent.

Changed in version 3.0: The *autoIndex* property has been renamed into *autoindex*.

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

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

`Table.`**`row`**
The associated Row instance (see *The Row class*).

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

Table methods - reading

`Table.`**`col`**`(`*name*`)`
Get a column from the table.

If a column called name exists in the table, it is read and returned as a NumPy object. If it does not exist, a KeyError is raised.

Examples

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

That statement is equivalent to:

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

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

`Table.iterrows(start=None, stop=None, step=None)`
Iterate over the table using a Row instance.

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

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

See also:

`tableextension.Row` the table row iterator and field accessor

Notes

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

Changed in version 3.0: If the *start* parameter is provided and *stop* is None then the table is iterated from *start* to the last line. In PyTables < 3.0 only one element was returned.

Examples

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

`Table.itersequence(sequence)`
Iterate over a sequence of row coordinates.

Notes

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

`Table.itersorted(sortby, checkCSI=False, start=None, stop=None, step=None)`
Iterate table data following the order of the index of sortby column.

The sortby column must have associated a full index. If you want to ensure a fully sorted order, the index must be a CSI one. You may want to use the checkCSI argument in order to explicitly check for the existence of a CSI index.

The meaning of the start, stop and step arguments is the same as in `Table.read()`.

Changed in version 3.0: If the *start* parameter is provided and *stop* is None then the table is iterated from *start* to the last line. In PyTables < 3.0 only one element was returned.

`Table.`**`read`**(*start=None*, *stop=None*, *step=None*, *field=None*, *out=None*)

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

The start, stop and step parameters can be used to select only a *range of rows* in the table. Their meanings are the same as in the built-in Python slices.

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

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

The out parameter may be used to specify a NumPy array to receive the output data. Note that the array must have the same size as the data selected with the other parameters. Note that the array's datatype is not checked and no type casting is performed, so if it does not match the datatype on disk, the output will not be correct.

When specifying a single nested column with the field parameter, and supplying an output buffer with the out parameter, the output buffer must contain all columns in the table. The data in all columns will be read into the output buffer. However, only the specified nested column will be returned from the method call.

When data is read from disk in NumPy format, the output will be in the current system's byteorder, regardless of how it is stored on disk. If the out parameter is specified, the output array also must be in the current system's byteorder.

Changed in version 3.0: Added the *out* parameter. Also the start, stop and step parameters now behave like in slice.

Examples

Reading the entire table:

```
t.read()
```

Reading record n. 6:

```
t.read(6, 7)
```

Reading from record n. 6 to the end of the table:

```
t.read(6)
```

`Table.`**`read_coordinates`**(*coords*, *field=None*)

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

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

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

`Table.`**`read_sorted`**(*sortby*, *checkCSI=False*, *field=None*, *start=None*, *stop=None*, *step=None*)

Read table data following the order of the index of sortby column.

The sortby column must have associated a full index. If you want to ensure a fully sorted order, the index must be a CSI one. You may want to use the checkCSI argument in order to explicitly check for the existence of a CSI index.

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

The meaning of the start, stop and step arguments is the same as in `Table.read()`.

Changed in version 3.0: The start, stop and step parameters now behave like in slice.

`Table.`**`__getitem__`**`(`*key*`)`

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

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

In addition, NumPy-style point selections are supported. In particular, if key is a list of row coordinates, the set of rows determined by it is returned. Furthermore, if key is an array of boolean values, only the coordinates where key is True are returned. Note that for the latter to work it is necessary that key list would contain exactly as many rows as the table has.

Examples

```
record = table[4]
recarray = table[4:1000:2]
recarray = table[[4,1000]]   # only retrieves rows 4 and 1000
recarray = table[[True, False, ..., True]]
```

Those statements are equivalent to:

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

Here, you can see how indexing can be used as a shorthand for the `Table.read()` and `Table.read_coordinates()` methods.

`Table.`**`__iter__`**`()`

Iterate over the table using a Row instance.

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

See also:

`tableextension.Row` the table row iterator and field accessor

Notes

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

Examples

```
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 ]
```

Table methods - writing

Table.**append**(*rows*)

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

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

Examples

```
import tables as tb

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

fileh = tb.open_file('test4.h5', mode='w')
table = fileh.create_table(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()
```

Table.**modify_column**(*start=None*, *stop=None*, *step=None*, *column=None*, *colname=None*)

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

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

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

Table.**modify_columns**(*start=None*, *stop=None*, *step=None*, *columns=None*, *names=None*)

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

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

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

Table.**modify_coordinates**(*coords*, *rows*)

Modify a series of rows in positions specified in coords.

The values in the selected rows will be modified with the data given in rows. This method returns the number of rows modified.

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

`Table.`**`modify_rows`**`(`*start=None*, *stop=None*, *step=None*, *rows=None*`)`

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

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

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

`Table.`**`remove_rows`**`(`*start=None*, *stop=None*, *step=None*`)`

Remove a range of rows in the table.

Changed in version 3.0: The start, stop and step parameters now behave like in slice.

See also:

remove_row()

Parameters **start** : int

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 : int

Sets the last row to be removed to stop-1, i.e. the end point is omitted (in the Python range() tradition). Negative values are also accepted.

step : int

The step size between rows to remove.

New in version 3.0.

Examples

Removing rows from 5 to 10 (excluded):

```
t.remove_rows(5, 10)
```

Removing all rows starting drom the 10th:

```
t.remove_rows(10)
```

Removing the 6th row:

```
t.remove_rows(6, 7)
```

Note: removing a single row can be done using the specific *`remove_row()`* method.

`Table.`**`remove_row`**`(`*n*`)`

Removes a row from the table.

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

Parameters **n** : int

The index of the row to remove.

.. versionadded:: 3.0 :

Table.**__setitem__**(*key*, *value*)

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

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

In addition, NumPy-style point selections are supported. In particular, if key is a list of row coordinates, the set of rows determined by it is set to value. Furthermore, if key is an array of boolean values, only the coordinates where key is True are set to values from value. Note that for the latter to work it is necessary that key list would contain exactly as many rows as the table has.

Examples

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

# Modify two existing rows
rows = numpy.rec.array([[457,'db1',1.2],[6,'de2',1.3]],
                       formats='i4,a3,f8')
table[1:30:2] = rows             # modify a table slice
table[[1,3]] = rows              # only modifies rows 1 and 3
table[[True,False,True]] = rows  # only modifies rows 0 and 2
```

Which is equivalent to:

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

Here, you can see how indexing can be used as a shorthand for the `Table.modify_rows()` and `Table.modify_coordinates()` methods.

Table methods - querying

Table.**get_where_list**(*condition*, *condvars=None*, *sort=False*, *start=None*, *stop=None*, *step=None*)

Get the row coordinates fulfilling the given condition.

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

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

Table.**read_where**(*condition*, *condvars=None*, *field=None*, *start=None*, *stop=None*, *step=None*)

Read table data fulfilling the given *condition*.

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

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

Table.**where**(*condition*, *condvars=None*, *start=None*, *stop=None*, *step=None*)

Iterate over values fulfilling a condition.

This method returns a Row iterator (see *The Row class*) which only selects rows in the table that satisfy the given condition (an expression-like string).

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

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

If a range is supplied (by setting some of the start, stop or step parameters), only the rows in that range and fulfilling the condition are used. The meaning of the start, stop and step parameters is the same as for Python slices.

When possible, indexed columns participating in the condition will be used to speed up the search. It is recommended that you place the indexed columns as left and out in the condition as possible. Anyway, this method has always better performance than regular Python selections on the table.

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

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

Examples

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

Note: A special care should be taken when the query condition includes string literals. Indeed Python 2 string literals are string of bytes while Python 3 strings are unicode objects.

Let's assume that the table `table` has the following structure:

```
class Record(IsDescription):
    col1 = StringCol(4)  # 4-character String of bytes
    col2 = IntCol()
    col3 = FloatCol()
```

The type of "col1" do not change depending on the Python version used (of course) and it always corresponds to strings of bytes.

Any condition involving "col1" should be written using the appropriate type for string literals in order to avoid `TypeErrors`.

The code below will work fine in Python 2 but will fail with a `TypeError` in Python 3:

```
condition = 'col1 == "AAAA"'
for record in table.where(condition):  # TypeError in Python3
    # do something with "record"
```

The reason is that in Python 3 "condition" implies a comparison between a string of bytes ("col1" contents) and an unicode literal ("AAAA").

The correct way to write the condition is:

```
condition = 'col1 == b"AAAA"'
```

Changed in version 3.0: The start, stop and step parameters now behave like in slice.

`Table.`**`append_where`**(*dstTable*, *condition*, *condvars=None*, *start=None*, *stop=None*, *step=None*)

Append rows fulfilling the condition to the dstTable table.

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

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

Changed in version 3.0: The *whereAppend* method has been renamed into *append_where*.

`Table.`**`will_query_use_indexing`**(*condition*, *condvars=None*)

Will a query for the condition use indexing?

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

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

Table methods - other

`Table.`**`copy`**(*newparent=None*, *newname=None*, *overwrite=False*, *createparents=False*, ***kwargs*)

Copy this table and return the new one.

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

Parameters **sortby** :

If specified, and sortby corresponds to a column with an index, then the copy will be sorted by this index. If you want to ensure a fully sorted order, the index must be a CSI one. A reverse sorted copy can be achieved by specifying a negative value for the step keyword. If sortby is omitted or None, the original table order is used.

checkCSI :

If true and a CSI index does not exist for the sortby column, an error will be raised. If false (the default), it does nothing. You can use this flag in order to explicitly check for the existence of a CSI index.

propindexes :

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

`Table.`**`flush_rows_to_index`**(*_lastrow=True*)

Add remaining rows in buffers to non-dirty indexes.

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

`Table.`**`get_enum`**`(`*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 *The Enum class*) is returned. If it is not of an enumerated type, a TypeError is raised. If the column does not exist, a KeyError is raised.

`Table.`**`reindex`**`()`

Recompute all the existing indexes in the table.

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

`Table.`**`reindex_dirty`**`()`

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

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

The Description class

class `tables.`**`Description`**`(`*classdict*, *nestedlvl=-1*, *validate=True*`)`

This class represents descriptions of the structure of tables.

An instance of this class is automatically bound to Table (see *The Table class*) objects when they are created. It provides a browseable representation of the structure of the table, made of non-nested (Col - see *The Col class and its descendants*) and nested (Description) columns.

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

Description attributes

`_v_colobjects`

A dictionary mapping the names of the columns hanging directly from the associated table or nested column to their respective descriptions (Col - see *The Col class and its descendants* or Description - see *The Description class* instances).

Changed in version 3.0: The *_v_colObjects* attobute has been renamed into *_v_colobjects*.

`_v_dflts`

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

`_v_dtype`

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

`_v_dtypes`

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

`_v_is_nested`

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

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

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

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

_v_nested_descr
A nested list of pairs of (name, format) tuples for all the columns under this table or nested column. You can use this as the dtype and descr arguments of NumPy array factories.

Changed in version 3.0: The *_v_nestedDescr* attribute has been renamed into *_v_nested_descr*.

_v_nested_formats
A nested list of the NumPy string formats (and shapes) of all the columns under this table or nested column. You can use this as the formats argument of NumPy array factories.

Changed in version 3.0: The *_v_nestedFormats* attribute has been renamed into *_v_nested_formats*.

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

_v_nested_names
A nested list of the names of all the columns under this table or nested column. You can use this as the names argument of NumPy array factories.

Changed in version 3.0: The *_v_nestedNames* attribute has been renamed into *_v_nested_names*.

_v_pathname
Pathname of the table or nested column.

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

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

Description methods

Description.**_f_walk**(*type='All'*)
Iterate over nested columns.

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

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

The Row class

class tables.tableextension.**Row**
Table row iterator and field accessor.

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

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

Row attributes

nrow

The current row number.

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

Row methods

Row.**append**()

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

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

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

Examples

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

Row.**fetch_all_fields**()

Retrieve all the fields in the current row.

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

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

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

Row.**update**()

Change the data of the current row in the dataset.

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

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

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

Examples

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

which modifies every tenth row in table. Or:

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

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

Row special methods

`Row.`**`__contains__`**`(`*item*`)`
A true value is returned if item is found in current row, false otherwise.

`Row.`**`__getitem__`**`(`*key*`)`
Get the row field specified by the *key*.

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

Examples

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

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

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

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

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

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

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

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

`Row.`**`__setitem__`**`(`*key*, *value*`)`
Set the key row field to the specified value.

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

Examples

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

which modifies every tenth row in the table.

The Cols class

class `tables.`**`Cols`**(*table*, *desc*)

Container for columns in a table or nested column.

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

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

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

Cols attributes

`_v_colnames`

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

`_v_colpathnames`

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

`_v_desc`

The associated Description instance (see *The Description class*).

Cols properties

`Cols.`**`_v_table`**

The parent Table instance (see *The Table class*).

Cols methods

Cols.**_f_col**(*colname*)

Get an accessor to the column colname.

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

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

Cols.**__getitem__**(*key*)

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

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

Examples

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

Those statements are equivalent to:

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

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

Cols.**__len__**()

Get the number of top level columns in table.

Cols.**__setitem__**(*key*, *value*)

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

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

Examples

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

Those statements are equivalent to:

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

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

The Column class

class tables.**Column**(*table*, *name*, *descr*)

Accessor for a non-nested column in a table.

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

Column attributes

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

name
The name of the associated column.

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

Parameters table :

The parent table instance

name :

The name of the column that is associated with this object

descr :

The parent description object

Column instance variables

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

Column.**index**
The Index instance (see *The Index class*) associated with this column (None if the column is not indexed).

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

Column.**maindim**
“The dimension along which iterators work. Its value is 0 (i.e. the first dimension).

Column.**shape**
The shape of this column.

Column.**table**
The parent Table instance (see *The Table class*).

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

Column methods

Column.**create_index**(*optlevel=6*, *kind='medium'*, *filters=None*, *tmp_dir=None*, *_blocksizes=None*, *_testmode=False*, *_verbose=False*)
Create an index for this column.

> **Warning:** In some situations it is useful to get a completely sorted index (CSI). For those cases, it is best to use the `Column.create_csindex()` method instead.

Parameters **optlevel** : int

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

kind : str

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

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

filters : Filters

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

tmp_dir :

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

`Column.`**`create_csindex`**(*filters=None*, *tmp_dir=None*, *_blocksizes=None*, *_testmode=False*, *_verbose=False*)

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

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

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

Notes

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

`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 you want to rebuild it.

This method does nothing if the column is not indexed.

Column.**reindex_dirty**()

Recompute the associated index only if it is dirty.

This can be useful when you have set *Table.autoindex* to false for the table and you want to update the column's index after an invalidating index operation (like *Table.remove_rows()*).

This method does nothing if the column is not indexed.

Column.**remove_index**()

Remove the index associated with this column.

This method does nothing if the column is not indexed. The removed index can be created again by calling the *Column.create_index()* method.

Column special methods

Column.**__getitem__**(*key*)

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

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

Examples

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

The output of this for a certain arbitrary table is:

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

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

Column.**__len__**()

Get the number of elements in the column.

This matches the length in rows of the parent table.

Column.**__setitem__**(*key*, *value*)

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

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

Examples

```
# 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.modify_columns(start=1, columns=[[-1]], names=['col1'])

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

4.5 Homogenous storage classes

4.5.1 The Array class

class tables.**Array**(*parentnode*, *name*, *obj=None*, *title=’‘*, *byteorder=None*, *_log=True*, *_atom=None*)

This class represents homogeneous datasets in an HDF5 file.

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

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

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

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*

name : str

The name of this node in its parent group.

obj :

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

Changed in version 3.0: Renamed form *object* into *obj*.

title :

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

byteorder :

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

Array instance variables

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

`Array.`**`rowsize`**
The size of the rows in bytes in dimensions orthogonal to *maindim*.

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

`Array.`**`nrows`**
The number of rows in the array.

Array methods

`Array.`**`get_enum`**`()`
Get the enumerated type associated with this array.

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

`Array.`**`iterrows`**`(`*start=None*, *stop=None*, *step=None*`)`
Iterate over the rows of the array.

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

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

Examples

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

Changed in version 3.0: If the *start* parameter is provided and *stop* is None then the array is iterated from *start* to the last line. In PyTables < 3.0 only one element was returned.

`Array.`**`next`**`()`
Get the next element of the array during an iteration.

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

`Array.`**`read`**(*start=None*, *stop=None*, *step=None*, *out=None*)

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

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

The out parameter may be used to specify a NumPy array to receive the output data. Note that the array must have the same size as the data selected with the other parameters. Note that the array's datatype is not checked and no type casting is performed, so if it does not match the datatype on disk, the output will not be correct. Also, this parameter is only valid when the array's flavor is set to 'numpy'. Otherwise, a TypeError will be raised.

When data is read from disk in NumPy format, the output will be in the current system's byteorder, regardless of how it is stored on disk. The exception is when an output buffer is supplied, in which case the output will be in the byteorder of that output buffer.

Changed in version 3.0: Added the *out* parameter.

Array special methods

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

`Array.`**`__getitem__`**(*key*)

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

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

Furthermore, NumPy-style fancy indexing, where a list of indices in a certain axis is specified, is also supported. Note that only one list per selection is supported right now. Finally, NumPy-style point and boolean selections are supported as well.

Examples

```
array1 = array[4]                       # simple selection
array2 = array[4:1000:2]                # slice selection
array3 = array[1, ..., ::2, 1:4, 4:]    # general slice selection
array4 = array[1, [1,5,10], ..., -1]    # fancy selection
array5 = array[np.where(array[:] > 4)]  # point selection
array6 = array[array[:] > 4]            # boolean selection
```

`Array.`**`__iter__`**()

Iterate over the rows of the array.

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

Examples

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

Which is equivalent to:

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

`Array.`**`__setitem__`**(*key*, *value*)

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

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

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

Furthermore, NumPy-style fancy indexing, where a list of indices in a certain axis is specified, is also supported. Note that only one list per selection is supported right now. Finally, NumPy-style point and boolean selections are supported as well.

Examples

```
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:] = numpy.arange(1728, shape=(4,3,2,4,3,6))
a6[1, [1,5,10], ..., -1] = arr    # fancy selection
a7[np.where(a6[:] > 4)] = 4       # point selection + broadcast
a8[arr > 4] = arr2                # boolean selection
```

4.5.2 The CArray class

class `tables.`**`CArray`**(*parentnode*, *name*, *atom=None*, *shape=None*, *title=’‘*, *filters=None*, *chunkshape=None*, *byteorder=None*, *_log=True*)

This class represents homogeneous datasets in an HDF5 file.

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

CArray includes all the instance variables and methods of Array. Only those with different behavior are mentioned here.

Parameters **parentnode** :

The parent *Group* object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

atom :

An *Atom* instance representing the *type* and *shape* of the atomic objects to be saved.

shape :

The shape of the new array.

title :

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

filters :

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

chunkshape :

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

byteorder :

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

Examples

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

```python
import numpy
import tables

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

h5f = tables.open_file(fileName, 'w')
ca = h5f.create_carray(h5f.root, 'carray', atom, shape,
                       filters=filters)

# Fill a hyperslab in ``ca``.
ca[10:60, 20:70] = numpy.ones((50, 50))
h5f.close()

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

The output for the previous script is something like:

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

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

4.5.3 The EArray class

class `tables.`**`EArray`**(*parentnode*, *name*, *atom=None*, *shape=None*, *title=''*, *filters=None*, *expectedrows=None*, *chunkshape=None*, *byteorder=None*, *_log=True*)

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

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

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

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

atom :

An *Atom* instance representing the *type* and *shape* of the atomic objects to be saved.

shape :

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

title :

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

filters :

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

expectedrows :

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

chunkshape :

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

byteorder :

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

Examples

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

```
import tables
import numpy

fileh = tables.open_file('earray1.h5', mode='w')
a = tables.StringAtom(itemsize=8)

# Use ``a`` as the object type for the enlargeable array.
array_c = fileh.create_earray(fileh.root, 'array_c', a, (0,),
                              "Chars")
array_c.append(numpy.array(['a'*2, 'b'*4], dtype='S8'))
array_c.append(numpy.array(['a'*6, 'b'*8, 'c'*10], dtype='S8'))

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

The output for the previous script is something like:

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

EArray methods

`EArray`.**append**(*sequence*)

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

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

4.5.4 The VLArray class

class `tables.`**VLArray**(*parentnode*, *name*, *atom=None*, *title=’‘*, *filters=None*, *expectedrows=None*, *chunkshape=None*, *byteorder=None*, *_log=True*)

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

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

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

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

Note: VLArray objects also support compression although compression is only performed on the data structures used internally by the HDF5 to take references of the location of the variable length data. Data itself (the raw data) are not compressed or filtered.

Please refer to the VLTypes Technical Note for more details on the topic.

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

atom :

An *Atom* instance representing the *type* and *shape* of the atomic objects to be saved.

title :

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

filters :

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

expectedrows :

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

New in version 3.0.

chunkshape :

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

byteorder :

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

.. versionchanged:: 3.0 :

The *expectedsizeinMB* parameter has been replaced by *expectedrows*.

Examples

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

```
import tables
from numpy import *

# Create a VLArray:
```

```
fileh = tables.open_file('vlarray1.h5', mode='w')
vlarray = fileh.create_vlarray(fileh.root, 'vlarray1',
tables.Int32Atom(shape=()),
                "ragged array of ints",
                filters=tables.Filters(1))

# Append some (variable length) rows:
vlarray.append(array([5, 6]))
vlarray.append(array([5, 6, 7]))
vlarray.append([5, 6, 9, 8])

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

# Now, do the same with native Python strings.
vlarray2 = fileh.create_vlarray(fileh.root, 'vlarray2',
tables.StringAtom(itemsize=2),
                    "ragged array of strings",
                    filters=tables.Filters(1))
vlarray2.flavor = 'python'

# Append some (variable length) rows:
print('-->', vlarray2.title)
vlarray2.append(['5', '66'])
vlarray2.append(['5', '6', '77'])
vlarray2.append(['5', '6', '9', '88'])

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

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

The output for the previous script is something like:

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

VLArray attributes

The instance variables below are provided in addition to those in Leaf (see *The Leaf class*).

atom

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

flavor

The type of data object read from this leaf.

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

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

nrows
The current number of rows in the array.

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

VLArray properties

VLArray.**size_on_disk**
The HDF5 library does not include a function to determine size_on_disk for variable-length arrays. Accessing this attribute will raise a NotImplementedError.

VLArray.**size_in_memory**
The size of this array's data in bytes when it is fully loaded into memory.

Note: When data is stored in a VLArray using the ObjectAtom type, it is first serialized using pickle, and then converted to a NumPy array suitable for storage in an HDF5 file. This attribute will return the size of that NumPy representation. If you wish to know the size of the Python objects after they are loaded from disk, you can use this ActiveState recipe.

VLArray methods

VLArray.**append**(*sequence*)
Add a sequence of data to the end of the dataset.

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

VLArray.**get_enum**()
Get the enumerated type associated with this array.

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

VLArray.**iterrows**(*start=None*, *stop=None*, *step=None*)
Iterate over the rows of the array.

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

If a range is not supplied, *all the rows* in the array are iterated upon. You can also use the *VLArray.__iter__()* special method for that purpose. If you only want to iterate over a given *range of rows* in the array, you may use the start, stop and step parameters.

Examples

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

Changed in version 3.0: If the *start* parameter is provided and *stop* is None then the array is iterated from *start* to the last line. In PyTables < 3.0 only one element was returned.

`VLArray.`**`next`**`()`

Get the next element of the array during an iteration.

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

`VLArray.`**`read`**`(`*start=None*, *stop=None*, *step=1*`)`

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

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

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

`VLArray.`**`get_row_size`**`()`

Return the total size in bytes of all the elements contained in a given row.

VLArray special methods

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

`VLArray.`**`__getitem__`**`(`*key*`)`

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

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

In addition, NumPy-style point selections are supported. In particular, if key is a list of row coordinates, the set of rows determined by it is returned. Furthermore, if key is an array of boolean values, only the coordinates where key is True are returned. Note that for the latter to work it is necessary that key list would contain exactly as many rows as the array has.

Examples

```
a_row = vlarray[4]
a_list = vlarray[4:1000:2]
a_list2 = vlarray[[0,2]]   # get list of coords
a_list3 = vlarray[[0,-2]]  # negative values accepted
a_list4 = vlarray[numpy.array([True,...,False])]  # array of bools
```

`VLArray.`**`__iter__`**`()`

Iterate over the rows of the array.

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

Examples

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

Which is equivalent to:

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

VLArray.**__setitem__**(*key*, *value*)

Set a row, or set of rows, in the array.

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

In addition, NumPy-style point selections are supported. In particular, if key is a list of row coordinates, the set of rows determined by it is set to value. Furthermore, if key is an array of boolean values, only the coordinates where key is True are set to values from value. Note that for the latter to work it is necessary that key list would contain exactly as many rows as the table has.

Note: When updating the rows of a VLArray object which uses a pseudo-atom, there is a problem: you can only update values with *exactly* the same size in bytes than the original row. This is very difficult to meet with object pseudo-atoms, because `pickle` applied on a Python object does not guarantee to return the same number of bytes than over another object, even if they are of the same class. This effectively limits the kinds of objects than can be updated in variable-length arrays.

Examples

```
vlarray[0] = vlarray[0] * 2 + 3
vlarray[99] = arange(96) * 2 + 3

# Negative values for the index are supported.
vlarray[-99] = vlarray[5] * 2 + 3
vlarray[1:30:2] = list_of_rows
vlarray[[1,3]] = new_1_and_3_rows
```

4.6 Link classes

4.6.1 The Link class

class tables.link.**Link**(*parentnode*, *name*, *target=None*, *_log=False*)

Abstract base class for all PyTables links.

A link is a node that refers to another node. The Link class inherits from Node class and the links that inherits from Link are SoftLink and ExternalLink. There is not a HardLink subclass because hard links behave like a regular Group or Leaf. Contrarily to other nodes, links cannot have HDF5 attributes. This is an HDF5 library limitation that might be solved in future releases.

See *Using links for more convenient access to nodes* for a small tutorial on how to work with links.

Link attributes

target

The path string to the pointed node.

Link instance variables

`Link._v_attrs`
A *NoAttrs* instance replacing the typical *AttributeSet* instance of other node objects. The purpose of *NoAttrs* is to make clear that HDF5 attributes are not supported in link nodes.

Link methods

The following methods are useful for copying, moving, renaming and removing links.

`Link.copy(`*newparent=None*, *newname=None*, *overwrite=False*, *createparents=False*`)`
Copy this link and return the new one.

See `Node._f_copy()` for a complete explanation of the arguments. Please note that there is no recursive flag since links do not have child nodes.

`Link.move(`*newparent=None*, *newname=None*, *overwrite=False*`)`
Move or rename this link.

See `Node._f_move()` for a complete explanation of the arguments.

`Link.remove()`
Remove this link from the hierarchy.

`Link.rename(`*newname=None*, *overwrite=False*`)`
Rename this link in place.

See `Node._f_rename()` for a complete explanation of the arguments.

4.6.2 The SoftLink class

class `tables.link.SoftLink(`*parentnode*, *name*, *target=None*, *_log=False*`)`
Represents a soft link (aka symbolic link).

A soft link is a reference to another node in the *same* file hierarchy. Provided that the target node exists, its attributes and methods can be accessed directly from the softlink using the normal . syntax.

Softlinks also have the following public methods/attributes:

- *target*
- *dereference()*
- *copy()*
- *move()*
- *remove()*
- *rename()*
- *is_dangling()*

Note that these will override any correspondingly named methods/attributes of the target node.

For backwards compatibility, it is also possible to obtain the target node via the *__call__()* special method (this action is called *dereferencing*; see below)

Examples

::

```
    >>> f = tables.open_file('/tmp/test_softlink.h5', 'w')
    >>> a = f.create_array('/', 'A', np.arange(10))
    >>> link_a = f.create_soft_link('/', 'link_A', target='/A')
```

# transparent read/write access to a softlinked node >>> link_a[0] = -1 >>> print(link_a[:], link_a.dtype) (array([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9]), dtype('int64'))

# dereferencing a softlink using the __call__() method >>> print(link_a() is a) True

# SoftLink.remove() overrides Array.remove() >>> link_a.remove() >>> print(link_a) <closed tables.link.SoftLink at 0x7febe97186e0> >>> print(a[:], a.dtype) (array([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9]), dtype('int64'))

SoftLink special methods

The following methods are specific for dereferrencing and representing soft links.

`SoftLink.`**`__call__`**`()`
Dereference *self.target* and return the object.

Examples

```
>>> f=tables.open_file('data/test.h5')
>>> print(f.root.link0)
/link0 (SoftLink) -> /another/path
>>> print(f.root.link0())
/another/path (Group) ''
```

`SoftLink.`**`__str__`**`()`
Return a short string representation of the link.

Examples

```
>>> f=tables.open_file('data/test.h5')
>>> print(f.root.link0)
/link0 (SoftLink) -> /path/to/node
```

4.6.3 The ExternalLink class

class `tables.link.`**`ExternalLink`**(*parentnode*, *name*, *target=None*, *_log=False*)
Represents an external link.

An external link is a reference to a node in *another* file. Getting access to the pointed node (this action is called *dereferencing*) is done via the `__call__()` special method (see below).

ExternalLink attributes

extfile
The external file handler, if the link has been dereferenced. In case the link has not been dereferenced yet, its value is None.

ExternalLink methods

ExternalLink.**umount**()
Safely unmount self.extfile, if opened.

ExternalLink special methods

The following methods are specific for dereferrencing and representing external links.

ExternalLink.**__call__**(***kwargs*)
Dereference self.target and return the object.

You can pass all the arguments supported by the *open_file()* function (except filename, of course) so as to open the referenced external file.

Examples

```
>>> f=tables.open_file('data1/test1.h5')
>>> print(f.root.link2)
/link2 (ExternalLink) -> data2/test2.h5:/path/to/node
>>> plink2 = f.root.link2('a')  # open in 'a'ppend mode
>>> print(plink2)
/path/to/node (Group) ''
>>> print(plink2._v_filename)
'data2/test2.h5'        # belongs to referenced file
```

ExternalLink.**__str__**()
Return a short string representation of the link.

Examples

```
>>> f=tables.open_file('data1/test1.h5')
>>> print(f.root.link2)
/link2 (ExternalLink) -> data2/test2.h5:/path/to/node
```

4.7 Declarative classes

In this section a series of classes that are meant to *declare* datatypes that are required for creating primary PyTables datasets are described.

4.7.1 The Atom class and its descendants

class `tables.`**`Atom`**(*nptype*, *shape*, *dflt*)

Defines the type of atomic cells stored in a dataset.

The meaning of *atomic* is that individual elements of a cell can not be extracted directly by indexing (i.e. __getitem__()) the dataset; e.g. if a dataset has shape (2, 2) and its atoms have shape (3,), to get the third element of the cell at (1, 0) one should use dataset[1,0][2] instead of dataset[1,0,2].

The Atom class is meant to declare the different properties of the *base element* (also known as *atom*) of CArray, EArray and VLArray datasets, although they are also used to describe the base elements of Array datasets. Atoms have the property that their length is always the same. However, you can grow datasets along the extensible dimension in the case of EArray or put a variable number of them on a VLArray row. Moreover, they are not restricted to scalar values, and they can be *fully multidimensional objects*.

Parameters **itemsize** : int

For types with a non-fixed size, this sets the size in bytes of individual items in the atom.

shape : tuple

Sets the shape of the atom. An integer shape of N is equivalent to the tuple (N,).

dflt :

Sets the default value for the atom.

The following are the public methods and attributes of the Atom class. :

Notes

A series of descendant classes are offered in order to make the use of these element descriptions easier. You should use a particular Atom descendant class whenever you know the exact type you will need when writing your code. Otherwise, you may use one of the Atom.from_*() factory Methods.

Atom attributes

`dflt`

The default value of the atom.

If the user does not supply a value for an element while filling a dataset, this default value will be written to disk. If the user supplies a scalar value for a multidimensional atom, this value is automatically *broadcast* to all the items in the atom cell. If dflt is not supplied, an appropriate zero value (or *null* string) will be chosen by default. Please note that default values are kept internally as NumPy objects.

`dtype`

The NumPy dtype that most closely matches this atom.

`itemsize`

Size in bytes of a single item in the atom. Specially useful for atoms of the string kind.

`kind`

The PyTables kind of the atom (a string).

`shape`

The shape of the atom (a tuple for scalar atoms).

`type`

The PyTables type of the atom (a string).

Atoms can be compared with atoms and other objects for strict (in)equality without having to compare individual attributes:

```
>>> atom1 = StringAtom(itemsize=10)  # same as ``atom2``
>>> atom2 = Atom.from_kind('string', 10)  # same as ``atom1``
>>> atom3 = IntAtom()
>>> atom1 == 'foo'
False
>>> atom1 == atom2
True
>>> atom2 != atom1
False
>>> atom1 == atom3
False
>>> atom3 != atom2
True
```

Atom properties

`Atom.`**`ndim`**
The number of dimensions of the atom.

New in version 2.4.

`Atom.`**`recarrtype`**
String type to be used in numpy.rec.array().

`Atom.`**`size`**
Total size in bytes of the atom.

Atom methods

`Atom.`**`copy`**`(`***override*`)`
Get a copy of the atom, possibly overriding some arguments.

Constructor arguments to be overridden must be passed as keyword arguments:

```
>>> atom1 = Int32Atom(shape=12)
>>> atom2 = atom1.copy()
>>> print(atom1)
Int32Atom(shape=(12,), dflt=0)
>>> print(atom2)
Int32Atom(shape=(12,), dflt=0)
>>> atom1 is atom2
False
>>> atom3 = atom1.copy(shape=(2, 2))
>>> print(atom3)
Int32Atom(shape=(2, 2), dflt=0)
>>> atom1.copy(foobar=42)
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'foobar'
```

Atom factory methods

classmethod `Atom.`**`from_dtype`**`(`*class_*, *dtype*, *dflt=None*`)`
Create an Atom from a NumPy dtype.

An optional default value may be specified as the dflt argument. Information in the dtype not represented in an Atom is ignored:

```
>>> import numpy
>>> Atom.from_dtype(numpy.dtype((numpy.int16, (2, 2))))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_dtype(numpy.dtype('Float64'))
Float64Atom(shape=(), dflt=0.0)
```

classmethod `Atom.from_kind`(*class_*, *kind*, *itemsize=None*, *shape=()*, *dflt=None*)
Create an Atom from a PyTables kind.

Optional item size, shape and default value may be specified as the itemsize, shape and dflt arguments, respectively. Bear in mind that not all atoms support a default item size:

```
>>> Atom.from_kind('int', itemsize=2, shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_kind('int', shape=(2, 2))
Int32Atom(shape=(2, 2), dflt=0)
>>> Atom.from_kind('int', shape=1)
Int32Atom(shape=(1,), dflt=0)
>>> Atom.from_kind('string', dflt=b'hello')
Traceback (most recent call last):
...
ValueError: no default item size for kind ``string``
>>> Atom.from_kind('Float')
Traceback (most recent call last):
...
ValueError: unknown kind: 'Float'
```

Moreover, some kinds with atypical constructor signatures are not supported; you need to use the proper constructor:

```
>>> Atom.from_kind('enum')
Traceback (most recent call last):
...
ValueError: the ``enum`` kind is not supported...
```

classmethod `Atom.from_sctype`(*class_*, *sctype*, *shape=()*, *dflt=None*)
Create an Atom from a NumPy scalar type sctype.

Optional shape and default value may be specified as the shape and dflt arguments, respectively. Information in the sctype not represented in an Atom is ignored:

```
>>> import numpy
>>> Atom.from_sctype(numpy.int16, shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_sctype('S5', dflt='hello')
Traceback (most recent call last):
...
ValueError: unknown NumPy scalar type: 'S5'
>>> Atom.from_sctype('Float64')
Float64Atom(shape=(), dflt=0.0)
```

classmethod `Atom.from_type`(*class_*, *type*, *shape=()*, *dflt=None*)
Create an Atom from a PyTables type.

Optional shape and default value may be specified as the shape and dflt arguments, respectively:

```
>>> Atom.from_type('bool')
BoolAtom(shape=(), dflt=False)
>>> Atom.from_type('int16', shape=(2, 2))
Int16Atom(shape=(2, 2), dflt=0)
>>> Atom.from_type('string40', dflt='hello')
Traceback (most recent call last):
...
ValueError: unknown type: 'string40'
>>> Atom.from_type('Float64')
Traceback (most recent call last):
...
ValueError: unknown type: 'Float64'
```

Atom Sub-classes

class `tables.`**`StringAtom`**(*itemsize*, *shape=()*, *dflt=’‘*)

Defines an atom of type string.

The item size is the *maximum* length in characters of strings.

`itemsize`

Size in bytes of a sigle item in the atom.

class `tables.`**`BoolAtom`**(*shape=()*, *dflt=False*)

Defines an atom of type bool.

class `tables.`**`IntAtom`**(*itemsize=4*, *shape=()*, *dflt=0*)

Defines an atom of a signed integral type (int kind).

class `tables.`**`Int8Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `int8`.

class `tables.`**`Int16Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `int16`.

class `tables.`**`Int32Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `int32`.

class `tables.`**`Int64Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `int64`.

class `tables.`**`UIntAtom`**(*itemsize=4*, *shape=()*, *dflt=0*)

Defines an atom of an unsigned integral type (uint kind).

class `tables.`**`UInt8Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `uint8`.

class `tables.`**`UInt16Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `uint16`.

class `tables.`**`UInt32Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `uint32`.

class `tables.`**`UInt64Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type `uint64`.

class `tables.`**`FloatAtom`**(*itemsize=8*, *shape=()*, *dflt=0.0*)

Defines an atom of a floating point type (float kind).

class `tables.`**`Float32Atom`**(*shape=()*, *dflt=0.0*)

Defines an atom of type `float32`.

class `tables.`**`Float64Atom`**(*shape=()*, *dflt=0.0*)

Defines an atom of type `float64`.

class `tables.`**`ComplexAtom`**(*itemsize*, *shape=()*, *dflt=0j*)

Defines an atom of kind complex.

Allowed item sizes are 8 (single precision) and 16 (double precision). This class must be used instead of more concrete ones to avoid confusions with numarray-like precision specifications used in PyTables 1.X.

`itemsize`

Size in bytes of a sigle item in the atom.

class `tables.`**`Time32Atom`**(*shape=()*, *dflt=0*)

Defines an atom of type time32.

class `tables.`**`Time64Atom`**(*shape=()*, *dflt=0.0*)

Defines an atom of type time64.

class `tables.`**`EnumAtom`**(*enum*, *dflt*, *base*, *shape=()*)

Description of an atom of an enumerated type.

Instances of this class describe the atom type used to store enumerated values. Those values belong to an enumerated type, defined by the first argument (enum) in the constructor of the atom, which accepts the same kinds of arguments as the Enum class (see *The Enum class*). The enumerated type is stored in the enum attribute of the atom.

A default value must be specified as the second argument (dflt) in the constructor; it must be the *name* (a string) of one of the enumerated values in the enumerated type. When the atom is created, the corresponding concrete value is broadcast and stored in the dflt attribute (setting different default values for items in a multidimensional atom is not supported yet). If the name does not match any value in the enumerated type, a KeyError is raised.

Another atom must be specified as the base argument in order to determine the base type used for storing the values of enumerated values in memory and disk. This *storage atom* is kept in the base attribute of the created atom. As a shorthand, you may specify a PyTables type instead of the storage atom, implying that this has a scalar shape.

The storage atom should be able to represent each and every concrete value in the enumeration. If it is not, a TypeError is raised. The default value of the storage atom is ignored.

The type attribute of enumerated atoms is always enum.

Enumerated atoms also support comparisons with other objects:

```
>>> enum = ['T0', 'T1', 'T2']
>>> atom1 = EnumAtom(enum, 'T0', 'int8')  # same as ``atom2``
>>> atom2 = EnumAtom(enum, 'T0', Int8Atom())  # same as ``atom1``
>>> atom3 = EnumAtom(enum, 'T0', 'int16')
>>> atom4 = Int8Atom()
>>> atom1 == enum
False
>>> atom1 == atom2
True
>>> atom2 != atom1
False
>>> atom1 == atom3
False
>>> atom1 == atom4
False
>>> atom4 != atom1
True
```

Examples

The next C enum construction:

```
enum myEnum {
    T0,
    T1,
    T2
};
```

would correspond to the following PyTables declaration:

```
>>> my_enum_atom = EnumAtom(['T0', 'T1', 'T2'], 'T0', 'int32')
```

Please note the dflt argument with a value of ‘T0’. Since the concrete value matching T0 is unknown right now (we have not used explicit concrete values), using the name is the only option left for defining a default value for the atom.

The chosen representation of values for this enumerated atom uses unsigned 32-bit integers, which surely wastes quite a lot of memory. Another size could be selected by using the base argument (this time with a full-blown storage atom):

```
>>> my_enum_atom = EnumAtom(['T0', 'T1', 'T2'], 'T0', UInt8Atom())
```

You can also define multidimensional arrays for data elements:

```
>>> my_enum_atom = EnumAtom(
...    ['T0', 'T1', 'T2'], 'T0', base='uint32', shape=(3,2))
```

for 3x2 arrays of uint32.

itemsize
Size in bytes of a sigle item in the atom.

Pseudo atoms

Now, there come three special classes, ObjectAtom, VLStringAtom and VLUnicodeAtom, that actually do not descend from Atom, but which goal is so similar that they should be described here. Pseudo-atoms can only be used with VLArray datasets (see *The VLArray class*), and they do not support multidimensional values, nor multiple values per row.

They can be recognised because they also have kind, type and shape attributes, but no size, itemsize or dflt ones. Instead, they have a base atom which defines the elements used for storage.

See `examples/vlarray1.py` and `examples/vlarray2.py` for further examples on VLArray datasets, including object serialization and string management.

ObjectAtom

class `tables.`**`ObjectAtom`**
Defines an atom of type object.

This class is meant to fit *any* kind of Python object in a row of a VLArray dataset by using pickle behind the scenes. Due to the fact that you can not foresee how long will be the output of the pickle serialization (i.e. the atom already has a *variable* length), you can only fit *one object per row*. However, you can still group several objects in a single tuple or list and pass it to the *`VLArray.append()`* method.

Object atoms do not accept parameters and they cause the reads of rows to always return Python objects. You can regard object atoms as an easy way to save an arbitrary number of generic Python objects in a VLArray dataset.

VLStringAtom

class `tables.`**`VLStringAtom`**

Defines an atom of type `vlstring`.

This class describes a *row* of the VLArray class, rather than an atom. It differs from the StringAtom class in that you can only add *one instance of it to one specific row*, i.e. the `VLArray.append()` method only accepts one object when the base atom is of this type.

Like StringAtom, this class does not make assumptions on the encoding of the string, and raw bytes are stored as is. Unicode strings are supported as long as no character is out of the ASCII set; otherwise, you will need to *explicitly* convert them to strings before you can save them. For full Unicode support, using VLUnicodeAtom (see *VLUnicodeAtom*) is recommended.

Variable-length string atoms do not accept parameters and they cause the reads of rows to always return Python strings. You can regard vlstring atoms as an easy way to save generic variable length strings.

VLUnicodeAtom

class `tables.`**`VLUnicodeAtom`**

Defines an atom of type vlunicode.

This class describes a *row* of the VLArray class, rather than an atom. It is very similar to VLStringAtom (see *VLStringAtom*), but it stores Unicode strings (using 32-bit characters a la UCS-4, so all strings of the same length also take up the same space).

This class does not make assumptions on the encoding of plain input strings. Plain strings are supported as long as no character is out of the ASCII set; otherwise, you will need to *explicitly* convert them to Unicode before you can save them.

Variable-length Unicode atoms do not accept parameters and they cause the reads of rows to always return Python Unicode strings. You can regard vlunicode atoms as an easy way to save variable length Unicode strings.

4.7.2 The Col class and its descendants

class `tables.`**`Col`**(*nptype*, *shape*, *dflt*)

Defines a non-nested column.

Col instances are used as a means to declare the different properties of a non-nested column in a table or nested column. Col classes are descendants of their equivalent Atom classes (see *The Atom class and its descendants*), but their instances have an additional _v_pos attribute that is used to decide the position of the column inside its parent table or nested column (see the IsDescription class in *The IsDescription class* for more information on column positions).

In the same fashion as Atom, you should use a particular Col descendant class whenever you know the exact type you will need when writing your code. Otherwise, you may use one of the Col.from_*() factory methods.

Each factory method inherited from the Atom class is available with the same signature, plus an additional pos parameter (placed in last position) which defaults to None and that may take an integer value. This parameter might be used to specify the position of the column in the table.

Besides, there are the next additional factory methods, available only for Col objects.

The following parameters are available for most Col-derived constructors.

Parameters **itemsize** : int

For types with a non-fixed size, this sets the size in bytes of individual items in the column.

shape : tuple

Sets the shape of the column. An integer shape of N is equivalent to the tuple (N,).

dflt :

Sets the default value for the column.

pos : int

Sets the position of column in table. If unspecified, the position will be randomly selected.

Col instance variables

In addition to the variables that they inherit from the Atom class, Col instances have the following attributes.

`Col.`**`_v_pos`**

The *relative* position of this column with regard to its column siblings.

Col factory methods

classmethod `Col.`**`from_atom`**(*class_*, *atom*, *pos=None*)

Create a Col definition from a PyTables atom.

An optional position may be specified as the pos argument.

Col sub-classes

class `tables.`**`StringCol`**(**args*, ***kwargs*)

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`BoolCol`**(**args*, ***kwargs*)

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`IntCol`**(**args*, ***kwargs*)

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Int8Col`**(**args*, ***kwargs*)

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Int16Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Int32Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Int64Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`UIntCol`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`UInt8Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`UInt16Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`UInt32Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`UInt64Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Float32Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Float64Col`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`ComplexCol`**`(`**args*, ***kwargs*`)`
Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`TimeCol`**`(`**args*, ***kwargs*`)`

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Time32Col`**`(`**args*, ***kwargs*`)`

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`Time64Col`**`(`**args*, ***kwargs*`)`

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

class `tables.`**`EnumCol`**`(`**args*, ***kwargs*`)`

Defines a non-nested column of a particular type.

The constructor accepts the same arguments as the equivalent *Atom* class, plus an additional `pos` argument for position information, which is assigned to the *_v_pos* attribute.

4.7.3 The IsDescription class

class `tables.`**`IsDescription`**

Description of the structure of a table or nested column.

This class is designed to be used as an easy, yet meaningful way to describe the structure of new Table (see *The Table class*) datasets or nested columns through the definition of *derived classes*. In order to define such a class, you must declare it as descendant of IsDescription, with as many attributes as columns you want in your table. The name of each attribute will become the name of a column, and its value will hold a description of it.

Ordinary columns can be described using instances of the Col class (see *The Col class and its descendants*). Nested columns can be described by using classes derived from IsDescription, instances of it, or name-description dictionaries. Derived classes can be declared in place (in which case the column takes the name of the class) or referenced by name.

Nested columns can have a _v_pos special attribute which sets the *relative* position of the column among sibling columns *also having explicit positions*. The pos constructor argument of Col instances is used for the same purpose. Columns with no explicit position will be placed afterwards in alphanumeric order.

Once you have created a description object, you can pass it to the Table constructor, where all the information it contains will be used to define the table structure.

IsDescription attributes

`_v_pos`

Sets the position of a possible nested column description among its sibling columns. This attribute can be specified *when declaring* an IsDescription subclass to complement its *metadata*.

`columns`

Maps the name of each column in the description to its own descriptive object. This attribute is *automatically created* when an IsDescription subclass is declared. Please note that declared columns can no longer be accessed as normal class variables after its creation.

4.7.4 Description helper functions

tables.description.**descr_from_dtype**(*dtype_*)

Get a description instance and byteorder from a (nested) NumPy dtype.

tables.description.**dtype_from_descr**(*descr*, *byteorder=None*)

Get a (nested) NumPy dtype from a description instance and byteorder.

The descr parameter can be a Description or IsDescription instance, sub-class of IsDescription or a dictionary.

4.7.5 The AttributeSet class

class tables.attributeset.**AttributeSet**(*node*)

Container for the HDF5 attributes of a Node.

This class provides methods to create new HDF5 node attributes, and to get, rename or delete existing ones.

Like in Group instances (see *The Group class*), AttributeSet instances make use of the *natural naming* convention, i.e. you can access the attributes on disk as if they were normal Python attributes of the AttributeSet instance.

This offers the user a very convenient way to access HDF5 node attributes. However, for this reason and in order not to pollute the object namespace, one can not assign *normal* attributes to AttributeSet instances, and their members use names which start by special prefixes as happens with Group objects.

Notes on native and pickled attributes

The values of most basic types are saved as HDF5 native data in the HDF5 file. This includes Python bool, int, float, complex and str (but not long nor unicode) values, as well as their NumPy scalar versions and homogeneous or *structured* NumPy arrays of them. When read, these values are always loaded as NumPy scalar or array objects, as needed.

For that reason, attributes in native HDF5 files will be always mapped into NumPy objects. Specifically, a multidimensional attribute will be mapped into a multidimensional ndarray and a scalar will be mapped into a NumPy scalar object (for example, a scalar H5T_NATIVE_LLONG will be read and returned as a numpy.int64 scalar).

However, other kinds of values are serialized using pickle, so you only will be able to correctly retrieve them using a Python-aware HDF5 library. Thus, if you want to save Python scalar values and make sure you are able to read them with generic HDF5 tools, you should make use of *scalar or homogeneous/structured array NumPy objects* (for example, numpy.int64(1) or numpy.array([1, 2, 3], dtype='int16')).

One more advice: because of the various potential difficulties in restoring a Python object stored in an attribute, you may end up getting a pickle string where a Python object is expected. If this is the case, you may wish to run pickle.loads() on that string to get an idea of where things went wrong, as shown in this example:

```
>>> import os, tempfile
>>> import tables
>>>
>>> class MyClass(object):
...   foo = 'bar'
...
>>> myObject = MyClass()  # save object of custom class in HDF5 attr
>>> h5fname = tempfile.mktemp(suffix='.h5')
>>> h5f = tables.open_file(h5fname, 'w')
>>> h5f.root._v_attrs.obj = myObject  # store the object
>>> print(h5f.root._v_attrs.obj.foo)  # retrieve it
```

```
bar
>>> h5f.close()
>>>
>>> del MyClass, myObject  # delete class of object and reopen file
>>> h5f = tables.open_file(h5fname, 'r')
>>> print(repr(h5f.root._v_attrs.obj))
'ccopy_reg\n_reconstructor...
>>> import pickle  # let's unpickle that to see what went wrong
>>> pickle.loads(h5f.root._v_attrs.obj)
Traceback (most recent call last):
...
AttributeError: 'module' object has no attribute 'MyClass'
>>> # So the problem was not in the stored object,
... # but in the *environment* where it was restored.
... h5f.close()
>>> os.remove(h5fname)
```

Notes on AttributeSet methods

Note that this class overrides the __getattr__(), __setattr__() and __delattr__() special methods. This allows you to read, assign or delete attributes on disk by just using the next constructs:

```
leaf.attrs.myattr = 'str attr'    # set a string (native support)
leaf.attrs.myattr2 = 3            # set an integer (native support)
leaf.attrs.myattr3 = [3, (1, 2)]  # a generic object (Pickled)
attrib = leaf.attrs.myattr        # get the attribute ``myattr``
del leaf.attrs.myattr             # delete the attribute ``myattr``
```

In addition, the dictionary-like __getitem__(), __setitem__() and __delitem__() methods are available, so you may write things like this:

```
for name in node._v_attrs._f_list():
    print("name: %s, value: %s" % (name, node._v_attrs[name]))
```

Use whatever idiom you prefer to access the attributes.

If an attribute is set on a target node that already has a large number of attributes, a PerformanceWarning will be issued.

AttributeSet attributes

_v_attrnames
A list with all attribute names.

_v_attrnamessys
A list with system attribute names.

_v_attrnamesuser
A list with user attribute names.

_v_unimplemented
A list of attribute names with unimplemented native HDF5 types.

AttributeSet properties

`AttributeSet.`**`_v_node`**

The *Node* instance this attribute set is associated with.

AttributeSet methods

`AttributeSet.`**`_f_copy`**`(`*where*`)`

Copy attributes to the where node.

Copies all user and certain system attributes to the given where node (a Node instance - see *The Node class*), replacing the existing ones.

`AttributeSet.`**`_f_list`**`(`*attrset='user'*`)`

Get a list of attribute names.

The attrset string selects the attribute set to be used. A 'user' value returns only user attributes (this is the default). A 'sys' value returns only system attributes. Finally, 'all' returns both system and user attributes.

`AttributeSet.`**`_f_rename`**`(`*oldattrname*, *newattrname*`)`

Rename an attribute from oldattrname to newattrname.

`AttributeSet.`**`__contains__`**`(`*name*`)`

Is there an attribute with that name?

A true value is returned if the attribute set has an attribute with the given name, false otherwise.

4.8 Helper classes

This section describes some classes that do not fit in any other section and that mainly serve for ancillary purposes.

4.8.1 The Filters class

class `tables.`**`Filters`**`(`*complevel=0*, *complib='zlib'*, *shuffle=True*, *fletcher32=False*, *least_significant_digit=None*, *_new=True*`)`

Container for filter properties.

This class is meant to serve as a container that keeps information about the filter properties associated with the chunked leaves, that is Table, CArray, EArray and VLArray.

Instances of this class can be directly compared for equality.

Parameters **complevel** : int

Specifies a compression level for data. The allowed range is 0-9. A value of 0 (the default) disables compression.

complib : str

Specifies the compression library to be used. Right now, 'zlib' (the default), 'lzo', 'bzip2' and 'blosc' are supported. Additional compressors for Blosc like 'blosc:blosclz' ('blosclz' is the default in case the additional compressor is not specified), 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy' and 'blosc:zlib' are supported too. Specifying a compression library which is not available in the system issues a FiltersWarning and sets the library to the default one.

shuffle : bool

Whether or not to use the *Shuffle* filter in the HDF5 library. This is normally used to improve the compression ratio. A false value disables shuffling and a true one enables it. The default value depends on whether compression is enabled or not; if compression is enabled, shuffling defaults to be enabled, else shuffling is disabled. Shuffling can only be used when compression is enabled.

fletcher32 : bool

Whether or not to use the *Fletcher32* filter in the HDF5 library. This is used to add a checksum on each data chunk. A false value (the default) disables the checksum.

least_significant_digit : int

If specified, data will be truncated (quantized). In conjunction with enabling compression, this produces 'lossy', but significantly more efficient compression. For example, if *least_significant_digit=1*, data will be quantized using `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). Default is *None*, or no quantization.

Note: quantization is only applied if some form of compression is enabled

Examples

This is a small example on using the Filters class:

```
import numpy
from tables import *

fileh = open_file('test5.h5', mode='w')
atom = Float32Atom()
filters = Filters(complevel=1, complib='blosc', fletcher32=True)
arr = fileh.create_earray(fileh.root, 'earray', atom, (0,2),
                         "A growable array", filters=filters)

# Append several rows in only one call
arr.append(numpy.array([[1., 2.],
                        [2., 3.],
                        [3., 4.]], dtype=numpy.float32))

# Print information on that enlargeable array
print("Result Array:")
print(repr(arr))
fileh.close()
```

This enforces the use of the Blosc library, a compression level of 1 and a Fletcher32 checksum filter as well. See the output of this example:

```
Result Array:
/earray (EArray(3, 2), fletcher32, shuffle, blosc(1)) 'A growable array'
type = float32
shape = (3, 2)
itemsize = 4
nrows = 3
extdim = 0
flavor = 'numpy'
byteorder = 'little'
```

Filters attributes

fletcher32
Whether the *Fletcher32* filter is active or not.

complevel
The compression level (0 disables compression).

complib
The compression filter used (irrelevant when compression is not enabled).

shuffle
Whether the *Shuffle* filter is active or not.

Filters methods

`Filters.`**`copy`**`(`***override*`)`

Get a copy of the filters, possibly overriding some arguments.

Constructor arguments to be overridden must be passed as keyword arguments.

Using this method is recommended over replacing the attributes of an instance, since instances of this class may become immutable in the future:

```
>>> filters1 = Filters()
>>> filters2 = filters1.copy()
>>> filters1 == filters2
True
>>> filters1 is filters2
False
>>> filters3 = filters1.copy(complevel=1)
Traceback (most recent call last):
...
ValueError: compression library ``None`` is not supported...
>>> filters3 = filters1.copy(complevel=1, complib='zlib')
>>> print(filters1)
Filters(complevel=0, shuffle=False, fletcher32=False, least_significant_digit=None)
>>> print(filters3)
Filters(complevel=1, complib='zlib', shuffle=False, fletcher32=False, least_significant_digit=No
>>> filters1.copy(foobar=42)
Traceback (most recent call last):
...
TypeError: __init__() got an unexpected keyword argument 'foobar'
```

4.8.2 The Index class

class `tables.index.`**`Index`**`(`*parentnode*, *name*, *atom=None*, *title='‘*, *kind=None*, *optlevel=None*, *filters=None*, *tmp_dir=None*, *expectedrows=0*, *byteorder=None*, *blocksizes=None*, *new=True*`)`

Represents the index of a column in a table.

This class is used to keep the indexing information for columns in a Table dataset (see *The Table class*). It is actually a descendant of the Group class (see *The Group class*), with some added functionality. An Index is always associated with one and only one column in the table.

Note: This class is mainly intended for internal use, but some of its documented attributes and methods may be interesting for the programmer.

Parameters **parentnode** :

The parent `Group` object.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

atom : Atom

An Atom object representing the shape and type of the atomic objects to be saved. Only scalar atoms are supported.

title :

Sets a TITLE attribute of the Index entity.

kind :

The desired kind for this index. The 'full' kind specifies a complete track of the row position (64-bit), while the 'medium', 'light' or 'ultralight' kinds only specify in which chunk the row is (using 32-bit, 16-bit and 8-bit respectively).

optlevel :

The desired optimization level for this index.

filters : Filters

An instance of the Filters class that provides information about the desired I/O filters to be applied during the life of this object.

tmp_dir :

The directory for the temporary files.

expectedrows :

Represents an user estimate about the number of row slices that will be added to the growable dimension in the IndexArray object.

byteorder :

The byteorder of the index datasets *on-disk*.

blocksizes :

The four main sizes of the compound blocks in index datasets (a low level parameter).

Index instance variables

`Index.`**`column`**

The Column (see *The Column class*) instance for the indexed column.

`Index.`**`dirty`**

Whether the index is dirty or not.

Dirty indexes are out of sync with column data, so they exist but they are not usable.

`Index.`**`filters`**

Filter properties for this index - see Filters in *The Filters class*.

`Index.`**`is_csi`**

Whether the index is completely sorted or not.

Changed in version 3.0: The *is_CSI* property has been renamed into *is_csi*.

`tables.index.Index.`**`nelements`**

The number of currently indexed rows for this column.

Index methods

`Index.`**`read_sorted`**`(`*start=None*, *stop=None*, *step=None*`)`

Return the sorted values of index in the specified range.

The meaning of the start, stop and step arguments is the same as in *`Table.read_sorted()`*.

`Index.`**`read_indices`**`(`*start=None*, *stop=None*, *step=None*`)`

Return the indices values of index in the specified range.

The meaning of the start, stop and step arguments is the same as in *`Table.read_sorted()`*.

Index special methods

`Index.`**`__getitem__`**`(`*key*`)`

Return the indices values of index in the specified range.

If key argument is an integer, the corresponding index is returned. If key is a slice, the range of indices determined by it is returned. A negative value of step in slice is supported, meaning that the results will be returned in reverse order.

This method is equivalent to `Index.read_indices()`.

4.8.3 The IndexArray class

class `tables.indexes.`**`IndexArray`**`(`*parentnode*, *name*, *atom=None*, *title=''*, *filters=None*, *byteorder=None*`)`

Represent the index (sorted or reverse index) dataset in HDF5 file.

All NumPy typecodes are supported except for complex datatypes.

Parameters **parentnode** :

The Index class from which this object will hang off.

Changed in version 3.0: Renamed from *parentNode* to *parentnode*.

name : str

The name of this node in its parent group.

atom :

An Atom object representing the shape and type of the atomic objects to be saved. Only scalar atoms are supported.

title :

Sets a TITLE attribute on the array entity.

filters : Filters

An instance of the Filters class that provides information about the desired I/O filters to be applied during the life of this object.

byteorder :

The byteroder of the data on-disk.

`chunksize`
The chunksize for this object.

`slicesize`
The slicesize for this object.

4.8.4 The Enum class

class `tables.misc.enum.`**`Enum`**`(`*enum*`)`
Enumerated type.

Each instance of this class represents an enumerated type. The values of the type must be declared *exhaustively* and named with *strings*, and they might be given explicit concrete values, though this is not compulsory. Once the type is defined, it can not be modified.

There are three ways of defining an enumerated type. Each one of them corresponds to the type of the only argument in the constructor of Enum:

- *Sequence of names*: each enumerated value is named using a string, and its order is determined by its position in the sequence; the concrete value is assigned automatically:

```
>>> boolEnum = Enum(['True', 'False'])
```

- *Mapping of names*: each enumerated value is named by a string and given an explicit concrete value. All of the concrete values must be different, or a ValueError will be raised:

```
>>> priority = Enum({'red': 20, 'orange': 10, 'green': 0})
>>> colors = Enum({'red': 1, 'blue': 1})
Traceback (most recent call last):
...
ValueError: enumerated values contain duplicate concrete values: 1
```

- *Enumerated type*: in that case, a copy of the original enumerated type is created. Both enumerated types are considered equal:

```
>>> prio2 = Enum(priority)
>>> priority == prio2
True
```

Please note that names starting with _ are not allowed, since they are reserved for internal usage:

```
>>> prio2 = Enum(['_xx'])
Traceback (most recent call last):
...
ValueError: name of enumerated value can not start with ``_``: '_xx'
```

The concrete value of an enumerated value is obtained by getting its name as an attribute of the Enum instance (see __getattr__()) or as an item (see __getitem__()). This allows comparisons between enumerated values and assigning them to ordinary Python variables:

```
>>> redv = priority.red
>>> redv == priority['red']
True
```

```
>>> redv > priority.green
True
>>> priority.red == priority.orange
False
```

The name of the enumerated value corresponding to a concrete value can also be obtained by using the __call__() method of the enumerated type. In this way you get the symbolic name to use it later with __getitem__():

```
>>> priority(redv)
'red'
>>> priority.red == priority[priority(priority.red)]
True
```

(If you ask, the __getitem__() method is not used for this purpose to avoid ambiguity in the case of using strings as concrete values.)

Enum special methods

Enum.**__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.

Examples

Let `enum` be an enumerated type defined as:

```
>>> enum = Enum({'T0': 0, 'T1': 2, 'T2': 5})
```

then:

```
>>> enum(5)
'T2'
>>> enum(42, None) is None
True
>>> enum(42)
Traceback (most recent call last):
  ...
ValueError: no enumerated value with that concrete value: 42
```

Enum.**__contains__**(*name*)

Is there an enumerated value with that name in the type?

If the enumerated type has an enumerated value with that name, True is returned. Otherwise, False is returned. The name must be a string.

This method does *not* check for concrete values matching a value in an enumerated type. For that, please use the `Enum.__call__()` method.

Examples

Let `enum` be an enumerated type defined as:

```
>>> enum = Enum({'T0': 0, 'T1': 2, 'T2': 5})
```

then:

```
>>> 'T1' in enum
True
>>> 'foo' in enum
False
>>> 0 in enum
Traceback (most recent call last):
  ...
TypeError: name of enumerated value is not a string: 0
>>> enum.T1 in enum  # Be careful with this!
Traceback (most recent call last):
  ...
TypeError: name of enumerated value is not a string: 2
```

Enum.**__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).

Examples

Let `enum*` be enumerated types defined as:

```
>>> enum1 = Enum({'T0': 0, 'T1': 2})
>>> enum2 = Enum(enum1)
>>> enum3 = Enum({'T1': 2, 'T0': 0})
>>> enum4 = Enum({'T0': 0, 'T1': 2, 'T2': 5})
>>> enum5 = Enum({'T0': 0})
>>> enum6 = Enum({'T0': 10, 'T1': 20})
```

then:

```
>>> enum1 == enum1
True
>>> enum1 == enum2 == enum3
True
>>> enum1 == enum4
False
>>> enum5 == enum1
False
>>> enum1 == enum6
False
```

Comparing enumerated types with other kinds of objects produces a false result:

```
>>> enum1 == {'T0': 0, 'T1': 2}
False
>>> enum1 == ['T0', 'T1']
False
>>> enum1 == 2
False
```

Enum.**__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.

Examples

Let `enum` be an enumerated type defined as:

```
>>> enum = Enum({'T0': 0, 'T1': 2, 'T2': 5})
```

then:

```
>>> enum.T1
2
>>> enum.foo
Traceback (most recent call last):
  ...
AttributeError: no enumerated value with that name: 'foo'
```

`Enum.`**`__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.

Examples

Let `enum` be an enumerated type defined as:

```
>>> enum = Enum({'T0': 0, 'T1': 2, 'T2': 5})
```

then:

```
>>> enum['T1']
2
>>> enum['foo']
Traceback (most recent call last):
  ...
KeyError: "no enumerated value with that name: 'foo'"
```

`Enum.`**`__iter__`**`()`

Iterate over the enumerated values.

Enumerated values are returned as (name, value) pairs *in no particular order*.

Examples

```
>>> enumvals = {'red': 4, 'green': 2, 'blue': 1}
>>> enum = Enum(enumvals)
>>> enumdict = dict([(name, value) for (name, value) in enum])
>>> enumvals == enumdict
True
```

`Enum.`**`__len__`**`()`

Return the number of enumerated values in the enumerated type.

Examples

```
>>> len(Enum(['e%d' % i for i in range(10)]))
10
```

Enum.**__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.

Examples

```
>>> repr(Enum({'name': 10}))
"Enum({'name': 10})"
```

4.8.5 The UnImplemented class

class tables.**UnImplemented**(*parentnode*, *name*)

This class represents datasets not supported by PyTables in an HDF5 file.

When reading a generic HDF5 file (i.e. one that has not been created with PyTables, but with some other HDF5 library based tool), chances are that the specific combination of datatypes or dataspaces in some dataset might not be supported by PyTables yet. In such a case, this dataset will be mapped into an UnImplemented instance and the user will still be able to access the complete object tree of the generic HDF5 file. The user will also be able to *read and write the attributes* of the dataset, *access some of its metadata*, and perform *certain hierarchy manipulation operations* like deleting or moving (but not copying) the node. Of course, the user will not be able to read the actual data on it.

This is an elegant way to allow users to work with generic HDF5 files despite the fact that some of its datasets are not supported by PyTables. However, if you are really interested in having full access to an unimplemented dataset, please get in contact with the developer team.

This class does not have any public instance variables or methods, except those inherited from the Leaf class (see *The Leaf class*).

byteorder = **None**

The endianness of data in memory ('big', 'little' or 'irrelevant').

nrows = **None**

The length of the first dimension of the data.

shape = **None**

The shape of the stored data.

4.8.6 The Unknown class

class tables.**Unknown**(*parentnode*, *name*)

This class represents nodes reported as *unknown* by the underlying HDF5 library.

This class does not have any public instance variables or methods, except those inherited from the Node class.

4.8.7 Exceptions module

In the exceptions module exceptions and warnings that are specific to PyTables are declared.

exception `tables.`**`HDF5ExtError`**(**args*, ***kargs*)

A low level HDF5 operation failed.

This exception is raised the low level PyTables components used for accessing HDF5 files. It usually signals that something is not going well in the HDF5 library or even at the Input/Output level.

Errors in the HDF5 C library may be accompanied by an extensive HDF5 back trace on standard error (see also `tables.silence_hdf5_messages()`).

Changed in version 2.4.

Parameters **message** :

error message

h5bt :

This parameter (keyword only) controls the HDF5 back trace handling. Any keyword arguments other than h5bt is ignored.

- if set to False the HDF5 back trace is ignored and the `HDF5ExtError.h5backtrace` attribute is set to None
- if set to True the back trace is retrieved from the HDF5 library and stored in the `HDF5ExtError.h5backtrace` attribute as a list of tuples
- if set to “VERBOSE” (default) the HDF5 back trace is stored in the `HDF5ExtError.h5backtrace` attribute and also included in the string representation of the exception
- if not set (or set to None) the default policy is used (see `HDF5ExtError.DEFAULT_H5_BACKTRACE_POLICY`)

`format_h5_backtrace`(*backtrace=None*)

Convert the HDF5 trace back represented as a list of tuples. (see `HDF5ExtError.h5backtrace`) into a string.

New in version 2.4.

`DEFAULT_H5_BACKTRACE_POLICY` **= ‘VERBOSE’**

Default policy for HDF5 backtrace handling

- if set to False the HDF5 back trace is ignored and the `HDF5ExtError.h5backtrace` attribute is set to None
- if set to True the back trace is retrieved from the HDF5 library and stored in the `HDF5ExtError.h5backtrace` attribute as a list of tuples
- if set to “VERBOSE” (default) the HDF5 back trace is stored in the `HDF5ExtError.h5backtrace` attribute and also included in the string representation of the exception

This parameter can be set using the `PT_DEFAULT_H5_BACKTRACE_POLICY` environment variable. Allowed values are “IGNORE” (or “FALSE”), “SAVE” (or “TRUE”) and “VERBOSE” to set the policy to False, True and “VERBOSE” respectively. The special value “DEFAULT” can be used to reset the policy to the default value

New in version 2.4.

`h5backtrace` **= None**

HDF5 back trace.

Contains the HDF5 back trace as a (possibly empty) list of tuples. Each tuple has the following format:

```
        (filename, line number, function name, text)
```

Depending on the value of the *h5bt* parameter passed to the initializer the h5backtrace attribute can be set to None. This means that the HDF5 back trace has been simply ignored (not retrieved from the HDF5 C library error stack) or that there has been an error (silently ignored) during the HDF5 back trace retrieval.

New in version 2.4.

See also:

`traceback.format_list` `traceback.format_list()`

exception `tables.`**`ClosedNodeError`**

The operation can not be completed because the node is closed.

For instance, listing the children of a closed group is not allowed.

exception `tables.`**`ClosedFileError`**

The operation can not be completed because the hosting file is closed.

For instance, getting an existing node from a closed file is not allowed.

exception `tables.`**`FileModeError`**

The operation can not be carried out because the mode in which the hosting file is opened is not adequate.

For instance, removing an existing leaf from a read-only file is not allowed.

exception `tables.`**`NodeError`**

Invalid hierarchy manipulation operation requested.

This exception is raised when the user requests an operation on the hierarchy which can not be run because of the current layout of the tree. This includes accessing nonexistent nodes, moving or copying or creating over an existing node, non-recursively removing groups with children, and other similarly invalid operations.

A node in a PyTables database cannot be simply overwritten by replacing it. Instead, the old node must be removed explicitely before another one can take its place. This is done to protect interactive users from inadvertedly deleting whole trees of data by a single erroneous command.

exception `tables.`**`NoSuchNodeError`**

An operation was requested on a node that does not exist.

This exception is raised when an operation gets a path name or a `(where, name)` pair leading to a nonexistent node.

exception `tables.`**`UndoRedoError`**

Problems with doing/redoing actions with Undo/Redo feature.

This exception indicates a problem related to the Undo/Redo mechanism, such as trying to undo or redo actions with this mechanism disabled, or going to a nonexistent mark.

exception `tables.`**`UndoRedoWarning`**

Issued when an action not supporting Undo/Redo is run.

This warning is only shown when the Undo/Redo mechanism is enabled.

exception `tables.`**`NaturalNameWarning`**

Issued when a non-pythonic name is given for a node.

This is not an error and may even be very useful in certain contexts, but one should be aware that such nodes cannot be accessed using natural naming (instead, `getattr()` must be used explicitly).

exception `tables.`**`PerformanceWarning`**

Warning for operations which may cause a performance drop.

This warning is issued when an operation is made on the database which may cause it to slow down on future operations (i.e. making the node tree grow too much).

exception `tables.`**`FlavorError`**

Unsupported or unavailable flavor or flavor conversion.

This exception is raised when an unsupported or unavailable flavor is given to a dataset, or when a conversion of data between two given flavors is not supported nor available.

exception `tables.`**`FlavorWarning`**

Unsupported or unavailable flavor conversion.

This warning is issued when a conversion of data between two given flavors is not supported nor available, and raising an error would render the data inaccessible (e.g. on a dataset of an unavailable flavor in a read-only file).

See the *FlavorError* class for more information.

exception `tables.`**`FiltersWarning`**

Unavailable filters.

This warning is issued when a valid filter is specified but it is not available in the system. It may mean that an available default filter is to be used instead.

exception `tables.`**`OldIndexWarning`**

Unsupported index format.

This warning is issued when an index in an unsupported format is found. The index will be marked as invalid and will behave as if doesn't exist.

exception `tables.`**`DataTypeWarning`**

Unsupported data type.

This warning is issued when an unsupported HDF5 data type is found (normally in a file created with other tool than PyTables).

exception `tables.`**`ExperimentalFeatureWarning`**

Generic warning for experimental features.

This warning is issued when using a functionality that is still experimental and that users have to use with care.

4.9 General purpose expression evaluator class

4.9.1 The Expr class

class `tables.`**`Expr`**(*expr*, *uservars=None*, ***kwargs*)

A class for evaluating expressions with arbitrary array-like objects.

Expr is a class for evaluating expressions containing array-like objects. With it, you can evaluate expressions (like "3 * a + 4 * b") that operate on arbitrary large arrays while optimizing the resources required to perform them (basically main memory and CPU cache memory). It is similar to the Numexpr package (see *[NUMEXPR]*), but in addition to NumPy objects, it also accepts disk-based homogeneous arrays, like the Array, CArray, EArray and Column PyTables objects.

All the internal computations are performed via the Numexpr package, so all the broadcast and upcasting rules of Numexpr applies here too. These rules are very similar to the NumPy ones, but with some exceptions due to the particularities of having to deal with potentially very large disk-based arrays. Be sure to read the documentation of the Expr constructor and methods as well as that of Numexpr, if you want to fully grasp these particularities.

Parameters **expr** : str

This specifies the expression to be evaluated, such as “2 * a + 3 * b”.

uservars : dict

This can be used to define the variable names appearing in *expr*. This mapping should consist of identifier-like strings pointing to any *Array*, *CArray*, *EArray*, *Column* or NumPy ndarray instances (or even others which will tried to be converted to ndarrays). When *uservars* is not provided or *None*, the current local and global namespace is sought instead of *uservars*. It is also possible to pass just some of the variables in expression via the *uservars* mapping, and the rest will be retrieved from the current local and global namespaces.

kwargs : dict

This is meant to pass additional parameters to the Numexpr kernel. This is basically the same as the kwargs argument in Numexpr.evaluate(), and is mainly meant for advanced use.

Examples

The following shows an example of using Expr.

```
>>> a = f.create_array('/', 'a', np.array([1,2,3]))
>>> b = f.create_array('/', 'b', np.array([3,4,5]))
>>> c = np.array([4,5,6])
>>> expr = tb.Expr("2 * a + b * c")    # initialize the expression
>>> expr.eval()                        # evaluate it
array([14, 24, 36])
>>> sum(expr)                          # use as an iterator
74
```

where you can see that you can mix different containers in the expression (whenever shapes are consistent).

You can also work with multidimensional arrays:

```
>>> a2 = f.create_array('/', 'a2', np.array([[1,2],[3,4]]))
>>> b2 = f.create_array('/', 'b2', np.array([[3,4],[5,6]]))
>>> c2 = np.array([4,5])           # This will be broadcasted
>>> expr = tb.Expr("2 * a2 + b2-c2")
>>> expr.eval()
array([[1, 3],
       [7, 9]])
>>> sum(expr)
array([ 8, 12])
```

Expr attributes

append_mode
The append mode for user-provided output containers.

maindim
Common main dimension for inputs in expression.

names
The names of variables in expression (list).

out
The user-provided container (if any) for the expression outcome.

o_start
The start range selection for the user-provided output.

o_stop
The stop range selection for the user-provided output.

o_step
The step range selection for the user-provided output.

shape
Common shape for the arrays in expression.

values
The values of variables in expression (list).

Expr methods

Expr.**eval**()

Evaluate the expression and return the outcome.

Because of performance reasons, the computation order tries to go along the common main dimension of all inputs. If not such a common main dimension is found, the iteration will go along the leading dimension instead.

For non-consistent shapes in inputs (i.e. shapes having a different number of dimensions), the regular NumPy broadcast rules applies. There is one exception to this rule though: when the dimensions orthogonal to the main dimension of the expression are consistent, but the main dimension itself differs among the inputs, then the shortest one is chosen for doing the computations. This is so because trying to expand very large on-disk arrays could be too expensive or simply not possible.

Also, the regular Numexpr casting rules (which are similar to those of NumPy, although you should check the Numexpr manual for the exceptions) are applied to determine the output type.

Finally, if the setOuput() method specifying a user container has already been called, the output is sent to this user-provided container. If not, a fresh NumPy container is returned instead.

> **Warning:** When dealing with large on-disk inputs, failing to specify an on-disk container may consume all your available memory.

Expr.**set_inputs_range**(*start=None*, *stop=None*, *step=None*)

Define a range for all inputs in expression.

The computation will only take place for the range defined by the start, stop and step parameters in the main dimension of inputs (or the leading one, if the object lacks the concept of main dimension, like a NumPy container). If not a common main dimension exists for all inputs, the leading dimension will be used instead.

Expr.**set_output**(*out*, *append_mode=False*)

Set out as container for output as well as the append_mode.

The out must be a container that is meant to keep the outcome of the expression. It should be an homogeneous type container and can typically be an Array, CArray, EArray, Column or a NumPy ndarray.

The append_mode specifies the way of which the output is filled. If true, the rows of the outcome are *appended* to the out container. Of course, for doing this it is necessary that out would have an append() method (like an EArray, for example).

If append_mode is false, the output is set via the __setitem__() method (see the Expr.set_output_range() for info on how to select the rows to be updated). If out is smaller than what is required by the expression, only the computations that are needed to fill up the container are carried out. If it is larger, the excess elements are unaffected.

`Expr.`**`set_output_range`**`(`*start=None*, *stop=None*, *step=None*`)`

Define a range for user-provided output object.

The output object will only be modified in the range specified by the start, stop and step parameters in the main dimension of output (or the leading one, if the object does not have the concept of main dimension, like a NumPy container).

Expr special methods

`Expr.`**`__iter__`**`()`

Iterate over the rows of the outcome of the expression.

This iterator always returns rows as NumPy objects, so a possible out container specified in *`Expr.set_output()`* method is ignored here.

4.10 Filenode Module

A file interface to nodes for PyTables databases.

The FileNode module provides a file interface for using inside of PyTables database files. Use the new_node() function to create a brand new file node which can be read and written as any ordinary Python file. Use the open_node() function to open an existing (i.e. created with new_node()) node for read-only or read-write access. Read acces is always available. Write access (enabled on new files and files opened with mode 'a+') only allows appending data to a file node.

Currently only binary I/O is supported.

See *filenode - simulating a filesystem with PyTables* for instructions on use.

Changed in version 3.0: In version 3.0 the module as been completely rewritten to be fully compliant with the interfaces defined in the `io` module.

4.10.1 Module constants

`tables.nodes.filenode.`**`NodeType`** **= 'file'**

Value for NODE_TYPE node system attribute.

`tables.nodes.filenode.`**`NodeTypeVersions`** **= [1, 2]**

Supported values for NODE_TYPE_VERSION node system attribute.

4.10.2 Module functions

`tables.nodes.filenode.`**`new_node`**`(`*h5file*, ***kwargs*`)`

Creates a new file node object in the specified PyTables file object.

Additional named arguments where and name must be passed to specify where the file node is to be created. Other named arguments such as title and filters may also be passed.

The special named argument expectedsize, indicating an estimate of the file size in bytes, may also be passed. It returns the file node object.

`tables.nodes.filenode.`**`open_node`**(*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.

`tables.nodes.filenode.`**`read_from_filenode`**(*h5file*, *filename*, *where*, *name=None*, *overwrite=False*, *create_target=False*)

Read a filenode from a PyTables file and write its contents to a file.

New in version 3.2.

Parameters **h5file** :

The PyTables file to be read from; can be either a string giving the file's location or a `File` object.

filename :

Path of the file where the contents of the filenode shall be written to. If *filename* points to a directory or ends with `/` (`\` on Windows), the filename will be set to the *_filename* (if present; otherwise the *name*) attribute of the read filenode.

where, name :

Location of the filenode where the data shall be read from. If no node *name* can be found at *where*, the first node at *where* whose *_filename* attribute matches *name* will be read.

overwrite :

Whether or not a possibly existing file of the specified *filename* shall be overwritten.

create_target :

Whether or not the folder hierarchy needed to accomodate the given target `filename` will be created.

`tables.nodes.filenode.`**`save_to_filenode`**(*h5file*, *filename*, *where*, *name=None*, *overwrite=False*, *title=''*, *filters=None*)

Save a file's contents to a filenode inside a PyTables file.

New in version 3.2.

Parameters **h5file** :

The PyTables file to be written to; can be either a string giving the file's location or a `File` object. If a file with name *h5file* already exists, it will be opened in mode `a`.

filename :

Path of the file which shall be stored within the PyTables file.

where, name :

Location of the filenode where the data shall be stored. If *name* is not given, and *where* is either a `Group` object or a string ending on `/`, the leaf name will be set to the file name of *filename*. The *name* will be modified to adhere to Python's natural naming convention; the original filename will be preserved in the filenode's *_filename* attribute.

overwrite :

Whether or not a possibly existing filenode of the specified name shall be overwritten.

title :

A description for this node (it sets the `TITLE` HDF5 attribute on disk).

filters :

An instance of the `Filters` class that provides information about the desired I/O filters to be applied during the life of this object.

4.10.3 The RawPyTablesIO base class

class `tables.nodes.filenode.`**`RawPyTablesIO`**`(`*node*, *mode=None*`)`

Base class for raw binary I/O on HDF5 files using PyTables.

RawPyTablesIO attributes

`RawPyTablesIO.`**`mode`**

File mode.

RawPyTablesIO methods

`RawPyTablesIO.`**`tell`**`()`

Return current stream position.

`RawPyTablesIO.`**`seek`**`(`*pos*, *whence=0*`)`

Change stream position.

Change the stream position to byte offset offset. offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

`RawPyTablesIO.`**`seekable`**`()`

Return whether object supports random access.

If False, seek(), tell() and truncate() will raise IOError. This method may need to do a test seek().

`RawPyTablesIO.`**`fileno`**`()`

Returns underlying file descriptor if one exists.

An IOError is raised if the IO object does not use a file descriptor.

`RawPyTablesIO.`**`close`**`()`

Flush and close the IO object.

This method has no effect if the file is already closed.

`RawPyTablesIO.`**`flush`**`()`

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

`RawPyTablesIO.`**`truncate`**`(`*pos=None*`)`

Truncate file to size bytes.

Size defaults to the current IO position as reported by tell(). Return the new size.

Currently, this method only makes sense to grow the file node, since data can not be rewritten nor deleted.

`RawPyTablesIO.`**`readable`**`()`

Return whether object was opened for reading.

If False, read() will raise IOError.

`RawPyTablesIO.`**`writable`**`()`

Return whether object was opened for writing.

If False, write() and truncate() will raise IOError.

`RawPyTablesIO.`**`readinto`**`(`*b*`)`

Read up to len(b) bytes into b.

Returns number of bytes read (0 for EOF), or None if the object is set not to block as has no data to read.

`RawPyTablesIO.`**`readline`**`(`*limit=-1*`)`

Read and return a line from the stream.

If limit is specified, at most limit bytes will be read.

The line terminator is always `\n` for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

`RawPyTablesIO.`**`write`**`(`*b*`)`

Write the given buffer to the IO stream.

Returns the number of bytes written, which may be less than len(b).

4.10.4 The ROFileNode class

class `tables.nodes.filenode.`**`ROFileNode`**`(`*node*`)`

Creates a new read-only file node.

Creates a new read-only file node associated with the specified PyTables node, providing a standard Python file interface to it. The node has to have been created on a previous occasion using the new_node() function.

The node used as storage is also made available via the read-only attribute node. Please do not tamper with this object if it's avoidable, since you may break the operation of the file node object.

The constructor is not intended to be used directly. Use the open_node() function in read-only mode ('r') instead.

Version 1 implements the file storage as a UInt8 uni-dimensional EArray.

Version 2 uses an UInt8 N vector EArray.

Changed in version 3.0: The offset attribute is no more available, please use seek/tell methods instead.

Changed in version 3.0: The line_separator property is no more available. The only line separator used for binary I/O is `\n`.

ROFileNode attributes

`ROFileNode.`**`attrs`**

A property pointing to the attribute set of the file node.

ROFileNode methods

`ROFileNode.`**`flush`**`()`

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

`ROFileNode.`**`read`**`()`

`ROFileNode.`**`readline`**`(`*limit=-1*`)`

Read and return a line from the stream.

If limit is specified, at most limit bytes will be read.

The line terminator is always `\n` for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

`ROFileNode.`**`readlines`**`()`

Return a list of lines from the stream.

hint can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds hint.

`ROFileNode.`**`close`**`()`

Flush and close the IO object.

This method has no effect if the file is already closed.

`ROFileNode.`**`seek`**`(`*pos*, *whence=0*`)`

Change stream position.

Change the stream position to byte offset offset. offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

`ROFileNode.`**`tell`**`()`

Return current stream position.

`ROFileNode.`**`readable`**`()`

Return whether object was opened for reading.

If False, read() will raise IOError.

`ROFileNode.`**`writable`**`()`

Return whether object was opened for writing.

If False, write() and truncate() will raise IOError.

`ROFileNode.`**`seekable`**`()`

Return whether object supports random access.

If False, seek(), tell() and truncate() will raise IOError. This method may need to do a test seek().

`ROFileNode.`**`fileno`**`()`

Returns underlying file descriptor if one exists.

An IOError is raised if the IO object does not use a file descriptor.

4.10.5 The RAFileNode class

class `tables.nodes.filenode.`**`RAFileNode`**(*node*, *h5file*, ***kwargs*)

Creates a new read-write file node.

The first syntax opens the specified PyTables node, while the second one creates a new node in the specified PyTables file. In the second case, 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.

Write access means reading as well as appending data is allowed.

The node used as storage is also made available via the read-only attribute node. Please do not tamper with this object if it's avoidable, since you may break the operation of the file node object.

The constructor is not intended to be used directly. Use the new_node() or open_node() functions instead.

Version 1 implements the file storage as a UInt8 uni-dimensional EArray.

Version 2 uses an UInt8 N vector EArray.

Changed in version 3.0: The offset attribute is no more available, please use seek/tell methods instead.

Changed in version 3.0: The line_separator property is no more available. The only line separator used for binary I/O is `\n`.

RAFileNode attributes

`RAFileNode.`**`attrs`**

A property pointing to the attribute set of the file node.

RAFileNode methods

`RAFileNode.`**`flush`**()

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

`RAFileNode.`**`read`**()

`RAFileNode.`**`readline`**(*limit=-1*)

Read and return a line from the stream.

If limit is specified, at most limit bytes will be read.

The line terminator is always `\n` for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

`RAFileNode.`**`readlines`**()

Return a list of lines from the stream.

hint can be specified to control the number of lines read: no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds hint.

`RAFileNode.`**`truncate`**(*pos=None*)

Truncate file to size bytes.

Size defaults to the current IO position as reported by tell(). Return the new size.

Currently, this method only makes sense to grow the file node, since data can not be rewritten nor deleted.

`RAFileNode.`**`write`**`(`*b*`)`

Write the given buffer to the IO stream.

Returns the number of bytes written, which may be less than len(b).

`RAFileNode.`**`writelines`**`()`

`RAFileNode.`**`close`**`()`

Flush and close the IO object.

This method has no effect if the file is already closed.

`RAFileNode.`**`seek`**`(`*pos*, *whence=0*`)`

Change stream position.

Change the stream position to byte offset offset. offset is interpreted relative to the position indicated by whence. Values for whence are:

- 0 – start of stream (the default); offset should be zero or positive
- 1 – current stream position; offset may be negative
- 2 – end of stream; offset is usually negative

Return the new absolute position.

`RAFileNode.`**`tell`**`()`

Return current stream position.

`RAFileNode.`**`readable`**`()`

Return whether object was opened for reading.

If False, read() will raise IOError.

`RAFileNode.`**`writable`**`()`

Return whether object was opened for writing.

If False, write() and truncate() will raise IOError.

`RAFileNode.`**`seekable`**`()`

Return whether object supports random access.

If False, seek(), tell() and truncate() will raise IOError. This method may need to do a test seek().

`RAFileNode.`**`fileno`**`()`

Returns underlying file descriptor if one exists.

An IOError is raised if the IO object does not use a file descriptor.

CHAPTER

FIVE

OPTIMIZATION TIPS

> ... durch planmässiges Tattonieren.
>
> [... through systematic, palpable experimentation.]
>
> —Johann Karl Friedrich Gauss [asked how he came upon his theorems]

On this chapter, you will get deeper knowledge of PyTables internals. PyTables has many tunable features so that you can improve the performance of your application. If you are planning to deal with really large data, you should read carefully this section in order to learn how to get an important efficiency boost for your code. But if your datasets are small (say, up to 10 MB) or your number of nodes is contained (up to 1000), you should not worry about that as the default parameters in PyTables are already tuned for those sizes (although you may want to adjust them further anyway). At any rate, reading this chapter will help you in your life with PyTables.

5.1 Understanding chunking

The underlying HDF5 library that is used by PyTables allows for certain datasets (the so-called *chunked* datasets) to take the data in bunches of a certain length, named *chunks*, and write them on disk as a whole, i.e. the HDF5 library treats chunks as atomic objects and disk I/O is always made in terms of complete chunks. This allows data filters to be defined by the application to perform tasks such as compression, encryption, check-summing, etc. on entire chunks.

HDF5 keeps a B-tree in memory that is used to map chunk structures on disk. The more chunks that are allocated for a dataset the larger the B-tree. Large B-trees take memory and cause file storage overhead as well as more disk I/O and higher contention forthe metadata cache. Consequently, it's important to balance between memory and I/O overhead (small B-trees) and time to access data (big B-trees).

In the next couple of sections, you will discover how to inform PyTables about the expected size of your datasets for allowing a sensible computation of the chunk sizes. Also, you will be presented some experiments so that you can get a feeling on the consequences of manually specifying the chunk size. Although doing this latter is only reserved to experienced people, these benchmarks may allow you to understand more deeply the chunk size implications and let you quickly start with the fine-tuning of this important parameter.

5.1.1 Informing PyTables about expected number of rows in tables or arrays

PyTables can determine a sensible chunk size to your dataset size if you helps it by providing an estimation of the final number of rows for an extensible leaf [1]. You should provide this information at leaf creation time by passing this value to the expectedrows argument of the `File.create_table()` method or `File.create_earray()` method (see *The EArray class*). For VLArray leaves, you must pass the expected size in MBytes by using the argument expectedsizein MB of `File.create_vlarray()` (see *The VLArray class*) instead.

[1] CArray nodes, though not extensible, are chunked and have their optimum chunk size automatically computed at creation time, since their final shape is known.

When your leaf size is bigger than 10 MB (take this figure only as a reference, not strictly), by providing this guess you will be optimizing the access to your data. When the table or array size is larger than, say 100MB, you are *strongly* suggested to provide such a guess; failing to do that may cause your application to do very slow I/O operations and to demand *huge* amounts of memory. You have been warned!

5.1.2 Fine-tuning the chunksize

> **Warning:** This section is mostly meant for experts. If you are a beginner, you must know that setting manually the chunksize is a potentially dangerous action.

Most of the time, informing PyTables about the extent of your dataset is enough. However, for more sophisticated applications, when one has special requirements for doing the I/O or when dealing with really large datasets, you should really understand the implications of the chunk size in order to be able to find the best value for your own application.

You can specify the chunksize for every chunked dataset in PyTables by passing the chunkshape argument to the corresponding constructors. It is important to point out that chunkshape is not exactly the same thing than a chunksize; in fact, the chunksize of a dataset can be computed multiplying all the dimensions of the chunkshape among them and multiplying the outcome by the size of the atom.

We are going to describe a series of experiments where an EArray of 15 GB is written with different chunksizes, and then it is accessed in both sequential (i.e. first element 0, then element 1 and so on and so forth until the data is exhausted) and random mode (i.e. single elements are read randomly all through the dataset). These benchmarks have been carried out with PyTables 2.1 on a machine with an Intel Core2 processor @ 3 GHz and a RAID-0 made of two SATA disks spinning at 7200 RPM, and using GNU/Linux with an XFS filesystem. The script used for the benchmarks is available in bench/optimal-chunksize.py.

In figures *Figure 1*, *Figure 2*, *Figure 3* and *Figure 4*, you can see how the chunksize affects different aspects, like creation time, file sizes, sequential read time and random read time. So, if you properly inform PyTables about the extent of your datasets, you will get an automatic chunksize value (256 KB in this case) that is pretty optimal for most of uses. However, if what you want is, for example, optimize the creation time when using the Zlib compressor, you may want to reduce the chunksize to 32 KB (see *Figure 1*). Or, if your goal is to optimize the sequential access time for an dataset compressed with Blosc, you may want to increase the chunksize to 512 KB (see *Figure 3*).

You will notice that, by manually specifying the chunksize of a leave you will not normally get a drastic increase in performance, but at least, you have the opportunity to fine-tune such an important parameter for improve performance.

Finally, it is worth noting that adjusting the chunksize can be specially important if you want to access your dataset by blocks of certain dimensions. In this case, it is normally a good idea to set your chunkshape to be the same than these dimensions; you only have to be careful to not end with a too small or too large chunksize. As always, experimenting prior to pass your application into production is your best ally.

5.2 Accelerating your searches

> **Note:** Many of the explanations and plots in this section and the forthcoming ones still need to be updated to include Blosc (see *[BLOSC]*), the new and powerful compressor added in PyTables 2.2 series. You should expect it to be the fastest compressor among all the described here, and its use is strongly recommended whenever you need extreme speed and not a very high compression ratio.

Searching in tables is one of the most common and time consuming operations that a typical user faces in the process of mining through his data. Being able to perform queries as fast as possible will allow more opportunities for finding

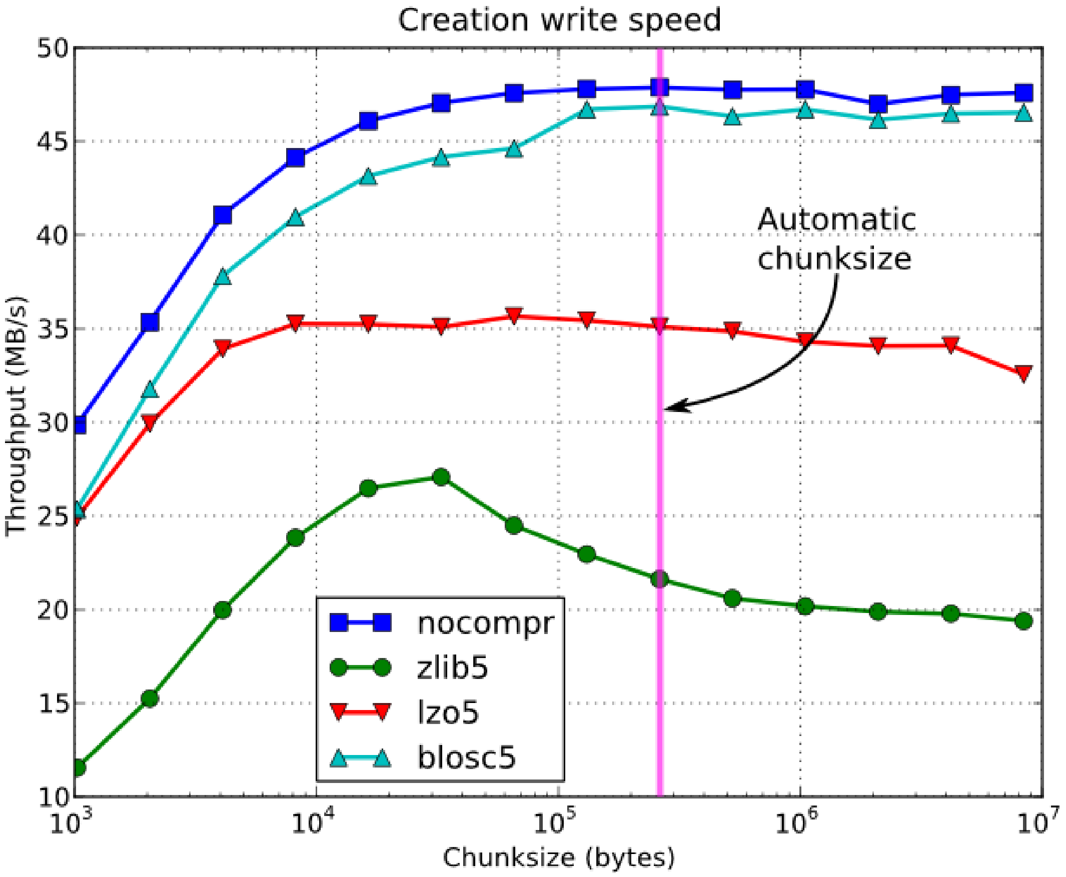


Fig. 5.1: **Figure 1. Creation time per element for a 15 GB EArray and different chunksizes.**

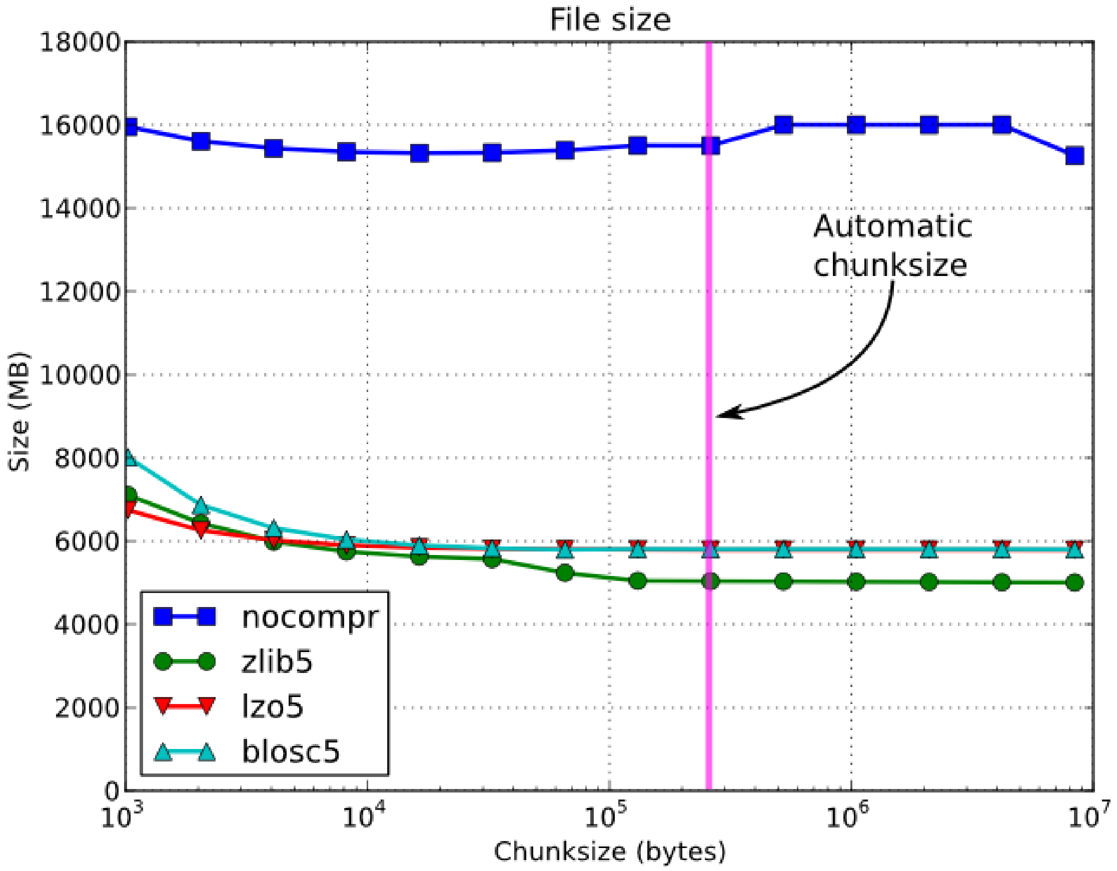


Fig. 5.2: **Figure 2. File sizes for a 15 GB EArray and different chunksizes.**

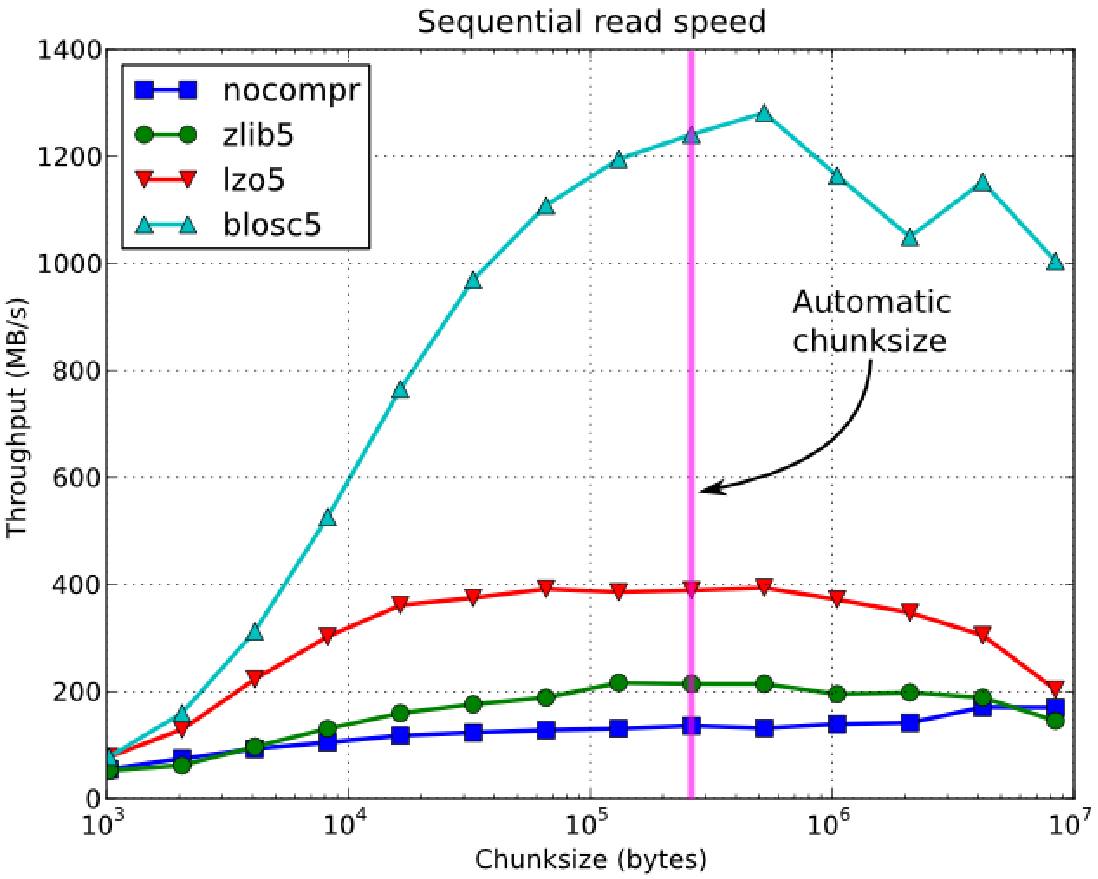


Fig. 5.3: **Figure 3. Sequential access time per element for a 15 GB EArray and different chunksizes.**

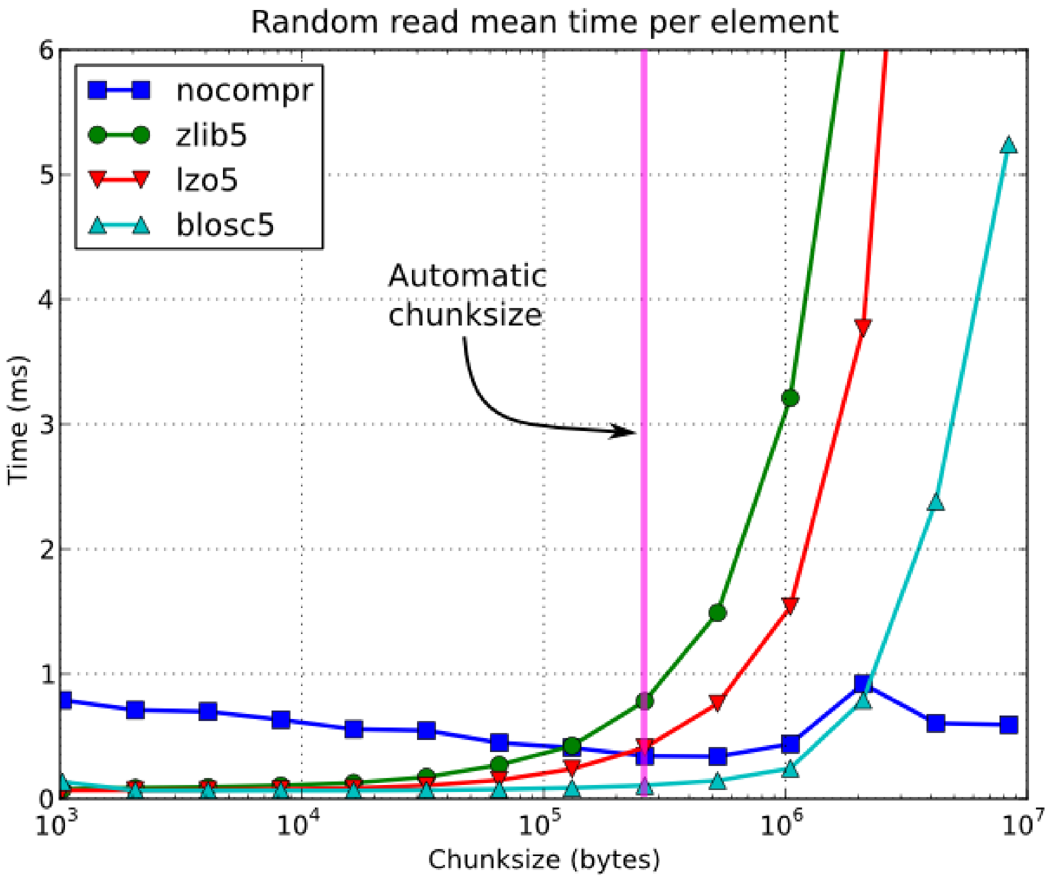


Fig. 5.4: **Figure 4. Random access time per element for a 15 GB EArray and different chunksizes.**

the desired information quicker and also allows to deal with larger datasets.

PyTables offers many sort of techniques so as to speed-up the search process as much as possible and, in order to give you hints to use them based, a series of benchmarks have been designed and carried out. All the results presented in this section have been obtained with synthetic, random data and using PyTables 2.1. Also, the tests have been conducted on a machine with an Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage (made of four spinning disks @ 7200 RPM), using GNU/Linux with an XFS filesystem. The script used for the benchmarks is available in bench/indexed_search.py. As your data, queries and platform may be totally different for your case, take this just as a guide because your mileage may vary (and will vary).

In order to be able to play with tables with a number of rows as large as possible, the record size has been chosen to be rather small (24 bytes). Here it is its definition:

```
class Record(tables.IsDescription):
    col1 = tables.Int32Col()
    col2 = tables.Int32Col()
    col3 = tables.Float64Col()
    col4 = tables.Float64Col()
```

In the next sections, we will be optimizing the times for a relatively complex query like this:

```
result = [row['col2'] for row in table if (
          ((row['col4'] >= lim1 and row['col4'] < lim2) or
          ((row['col2'] > lim3 and row['col2'] < lim4])) and
          ((row['col1']+3.1*row['col2']+row['col3']*row['col4']) > lim5)
          )]
```

(for future reference, we will call this sort of queries *regular* queries). So, if you want to see how to greatly improve the time taken to run queries like this, keep reading.

5.2.1 In-kernel searches

PyTables provides a way to accelerate data selections inside of a single table, through the use of the *Table methods - querying* iterator and related query methods. This mode of selecting data is called *in-kernel*. Let’s see an example of an *in-kernel* query based on the *regular* one mentioned above:

```
result = [row['col2'] for row in table.where(
            '''(((col4 >= lim1) & (col4 < lim2)) |
               ((col2 > lim3) & (col2 < lim4)) &
               ((col1+3.1*col2+col3*col4) > lim5))''')]
```

This simple change of mode selection can improve search times quite a lot and actually make PyTables very competitive when compared against typical relational databases as you can see in *Figure 5* and *Figure 6*.

By looking at *Figure 5* you can see how in the case that table data fits easily in memory, in-kernel searches on uncompressed tables are generally much faster (10x) than standard queries as well as PostgreSQL (5x). Regarding compression, we can see how Zlib compressor actually slows down the performance of in-kernel queries by a factor 3.5x; however, it remains faster than PostgreSQL (40%). On his hand, LZO compressor only decreases the performance by a 75% with respect to uncompressed in-kernel queries and is still a lot faster than PostgreSQL (3x). Finally, one can observe that, for low selectivity queries (large number of hits), PostgreSQL performance degrades quite steadily, while in PyTables this slow down rate is significantly smaller. The reason of this behaviour is not entirely clear to the authors, but the fact is clearly reproducible in our benchmarks.

But, why in-kernel queries are so fast when compared with regular ones?. The answer is that in regular selection mode the data for all the rows in table has to be brought into Python space so as to evaluate the condition and decide if the corresponding field should be added to the result list. On the contrary, in the in-kernel mode, the condition is passed to the PyTables kernel (hence the name), written in C, and evaluated there at full C speed (with the help of the integrated Numexpr package, see *[NUMEXPR]*), so that the only values that are brought to Python space are the rows

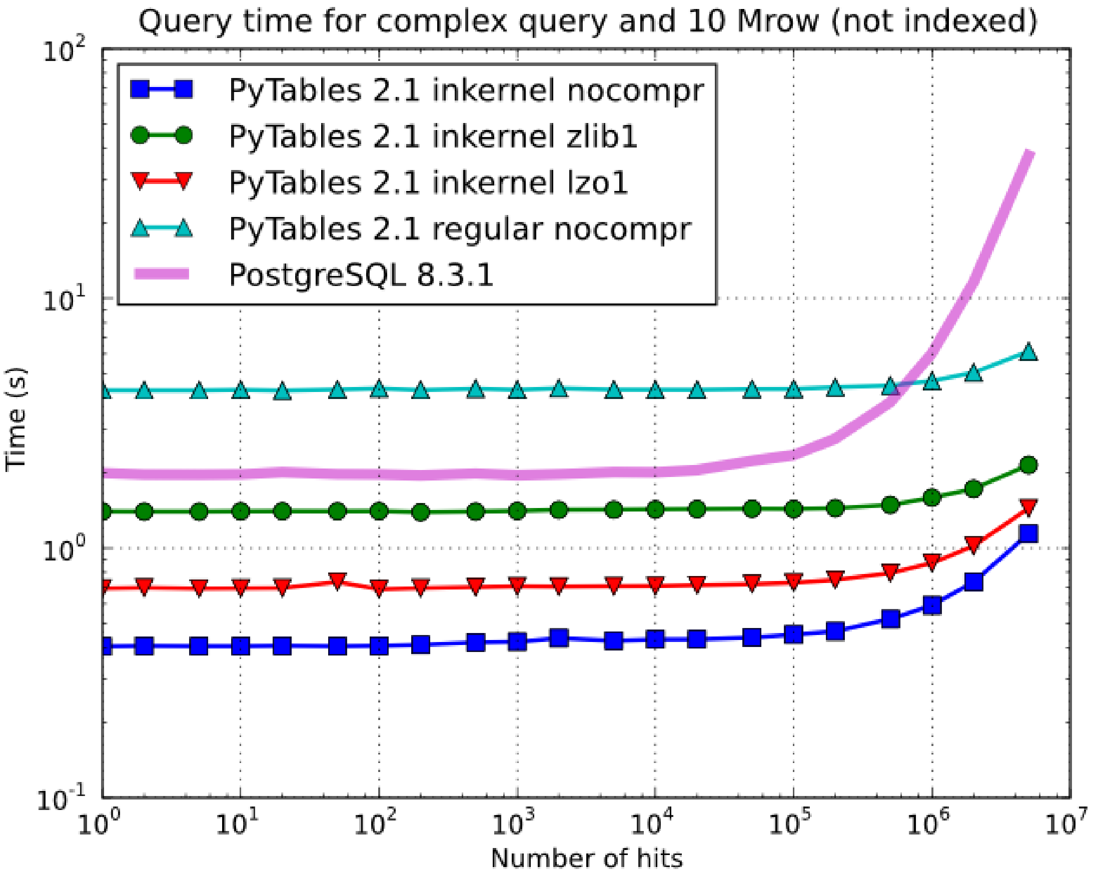


Fig. 5.5: **Figure 5. Times for non-indexed complex queries in a small table with 10 millions of rows: the data fits in memory.**

that fulfilled the condition. Hence, for selections that only have a relatively small number of hits (compared with the total amount of rows), the savings are very large. It is also interesting to note the fact that, although for queries with a large number of hits the speed-up is not as high, it is still very important.

On the other hand, when the table is too large to fit in memory (see *Figure 6*), the difference in speed between regular and in-kernel is not so important, but still significant (2x). Also, and curiously enough, large tables compressed with Zlib offers slightly better performance (around 20%) than uncompressed ones; this is because the additional CPU spent by the uncompressor is compensated by the savings in terms of net I/O (one has to read less actual data from disk). However, when using the extremely fast LZO compressor, it gives a clear advantage over Zlib, and is up to 2.5x faster than not using compression at all. The reason is that LZO decompression speed is much faster than Zlib, and that allows PyTables to read the data at full disk speed (i.e. the bottleneck is in the I/O subsystem, not in the CPU). In this case the compression rate is around 2.5x, and this is why the data can be read 2.5x faster. So, in general, using the LZO compressor is the best way to ensure best reading/querying performance for out-of-core datasets (more about how compression affects performance in *Compression issues*).

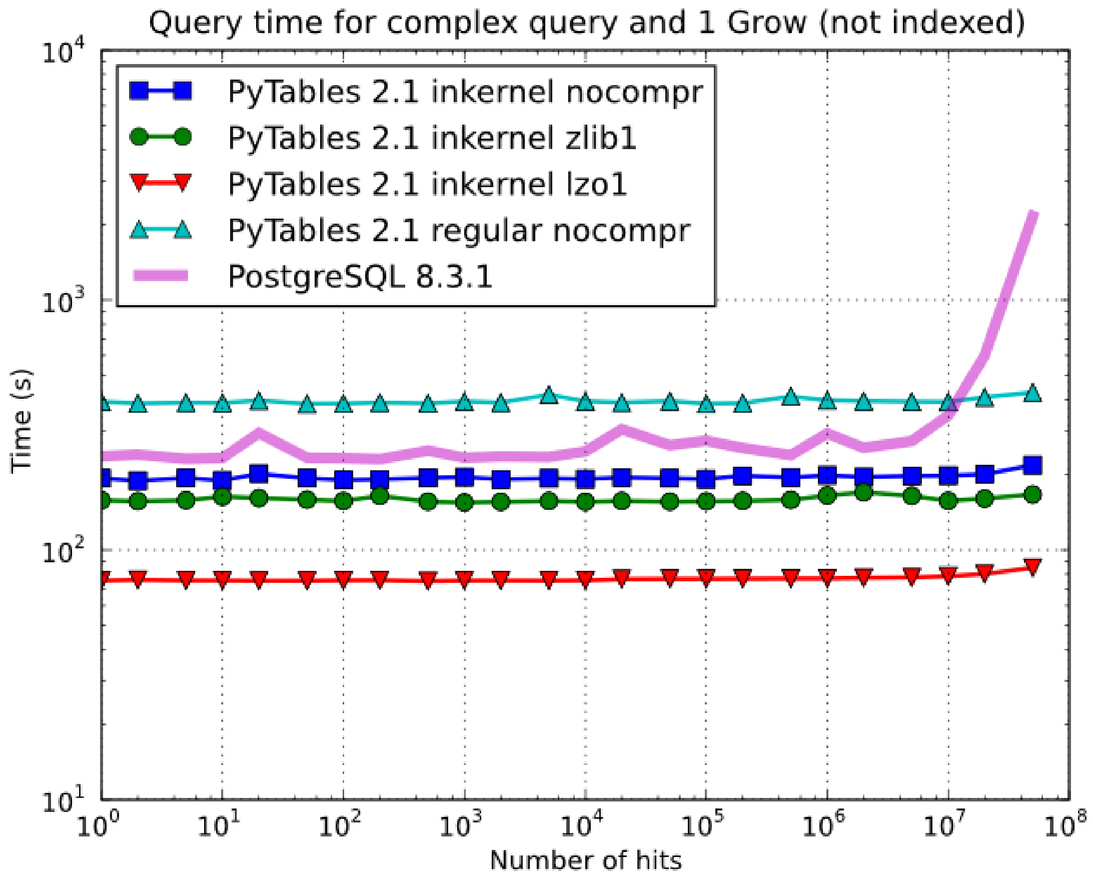


Fig. 5.6: **Figure 6. Times for non-indexed complex queries in a large table with 1 billion of rows: the data does not fit in memory.**

Furthermore, you can mix the *in-kernel* and *regular* selection modes for evaluating arbitrarily complex conditions making use of external functions. Look at this example:

```
result = [ row['var2']
           for row in table.where('(var3 == "foo") & (var1 <= 20)')
           if your_function(row['var2']) ]
```

Here, we use an *in-kernel* selection to choose rows according to the values of the var3 and var1 fields. Then, we apply a *regular* selection to complete the query. Of course, when you mix the *in-kernel* and *regular* selection modes you should pass the most restrictive condition to the *in-kernel* part, i.e. to the where() iterator. In situations where it is not clear which is the most restrictive condition, you might want to experiment a bit in order to find the best combination.

However, since in-kernel condition strings allow rich expressions allowing the coexistence of multiple columns, variables, arithmetic operations and many typical functions, it is unlikely that you will be forced to use external regular selections in conditions of small to medium complexity. See *Condition Syntax* for more information on in-kernel condition syntax.

5.2.2 Indexed searches

When you need more speed than *in-kernel* selections can offer you, PyTables offers a third selection method, the so-called *indexed* mode (based on the highly efficient OPSI indexing engine). In this mode, you have to decide which column(s) you are going to apply your selections over, and index them. Indexing is just a kind of sorting operation over a column, so that searches along such a column (or columns) will look at this sorted information by using a *binary search* which is much faster than the *sequential search* described in the previous section.

You can index the columns you want by calling the `Column.create_index()` method on an already created table. For example:

```
indexrows = table.cols.var1.create_index()
indexrows = table.cols.var2.create_index()
indexrows = table.cols.var3.create_index()
```

will create indexes for all var1, var2 and var3 columns.

After you have indexed a series of columns, the PyTables query optimizer will try hard to discover the usable indexes in a potentially complex expression. However, there are still places where it cannot determine that an index can be used. See below for examples where the optimizer can safely determine if an index, or series of indexes, can be used or not.

Example conditions where an index can be used:

- var1 >= “foo” (var1 is used)
- var1 >= mystr (var1 is used)
- (var1 >= “foo”) & (var4 > 0.0) (var1 is used)
- (“bar” <= var1) & (var1 < “foo”) (var1 is used)
- ((“bar” <= var1) & (var1 < “foo”)) & (var4 > 0.0) (var1 is used)
- (var1 >= “foo”) & (var3 > 10) (var1 and var3 are used)
- (var1 >= “foo”) | (var3 > 10) (var1 and var3 are used)
- ~(var1 >= “foo”) | ~(var3 > 10) (var1 and var3 are used)

Example conditions where an index can *not* be used:

- var4 > 0.0 (var4 is not indexed)
- var1 != 0.0 (range has two pieces)
- ~((“bar” <= var1) & (var1 < “foo”)) & (var4 > 0.0) (negation of a complex boolean expression)

Note: From PyTables 2.3 on, several indexes can be used in a single query.

Note: If you want to know for sure whether a particular query will use indexing or not (without actually running it), you are advised to use the `Table.will_query_use_indexing()` method.

One important aspect of the new indexing in PyTables (>= 2.3) is that it has been designed from the ground up with the goal of being capable to effectively manage very large tables. To this goal, it sports a wide spectrum of different quality levels (also called optimization levels) for its indexes so that the user can choose the best one that suits her needs (more or less size, more or less performance).

In *Figure 7*, you can see that the times to index columns in tables can be really short. In particular, the time to index a column with 1 billion rows (1 Gigarow) with the lowest optimization level is less than 4 minutes while indexing the same column with full optimization (so as to get a completely sorted index or CSI) requires around 1 hour. These are rather competitive figures compared with a relational database (in this case, PostgreSQL 8.3.1, which takes around 1.5 hours for getting the index done). This is because PyTables is geared towards read-only or append-only tables and takes advantage of this fact to optimize the indexes properly. On the contrary, most relational databases have to deliver decent performance in other scenarios as well (specially updates and deletions), and this fact leads not only to slower index creation times, but also to indexes taking much more space on disk, as you can see in *Figure 8*.

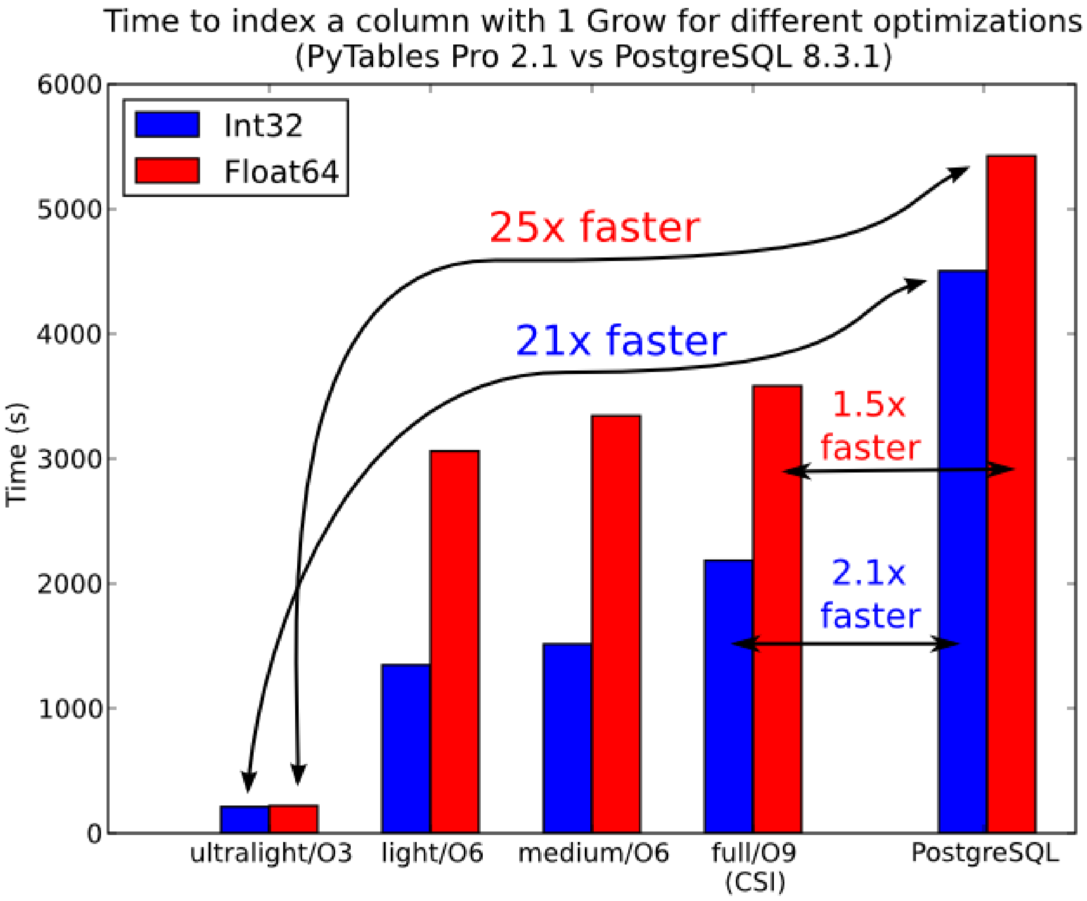


Fig. 5.7: **Figure 7. Times for indexing an Int32 and Float64 column.**

The user can select the index quality by passing the desired optlevel and kind arguments to the `Column.create_index()` method. We can see in figures *Figure 7* and *Figure 8* how the different optimization levels affects index time creation and index sizes.

So, which is the effect of the different optimization levels in terms of query times? You can see that in *Figure 9*.

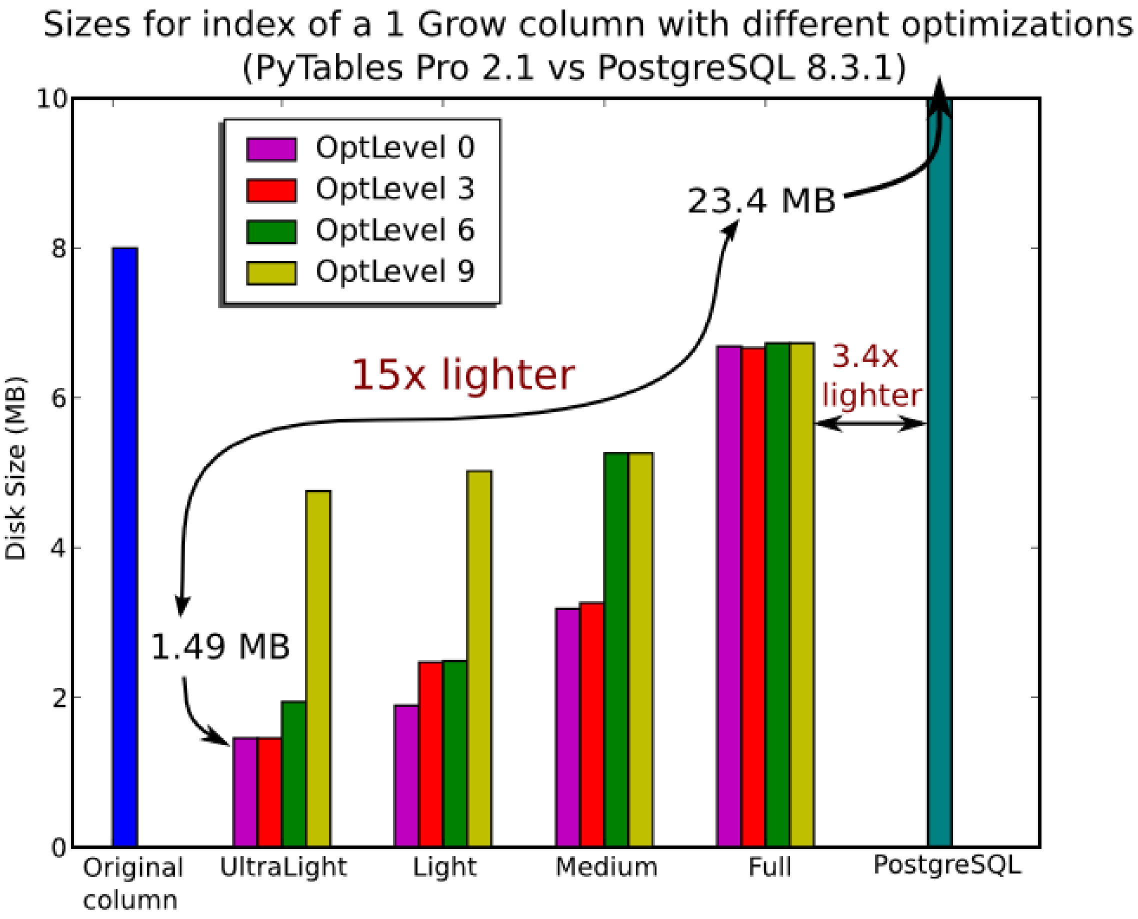


Fig. 5.8: **Figure 8. Sizes for an index of a Float64 column with 1 billion of rows.**

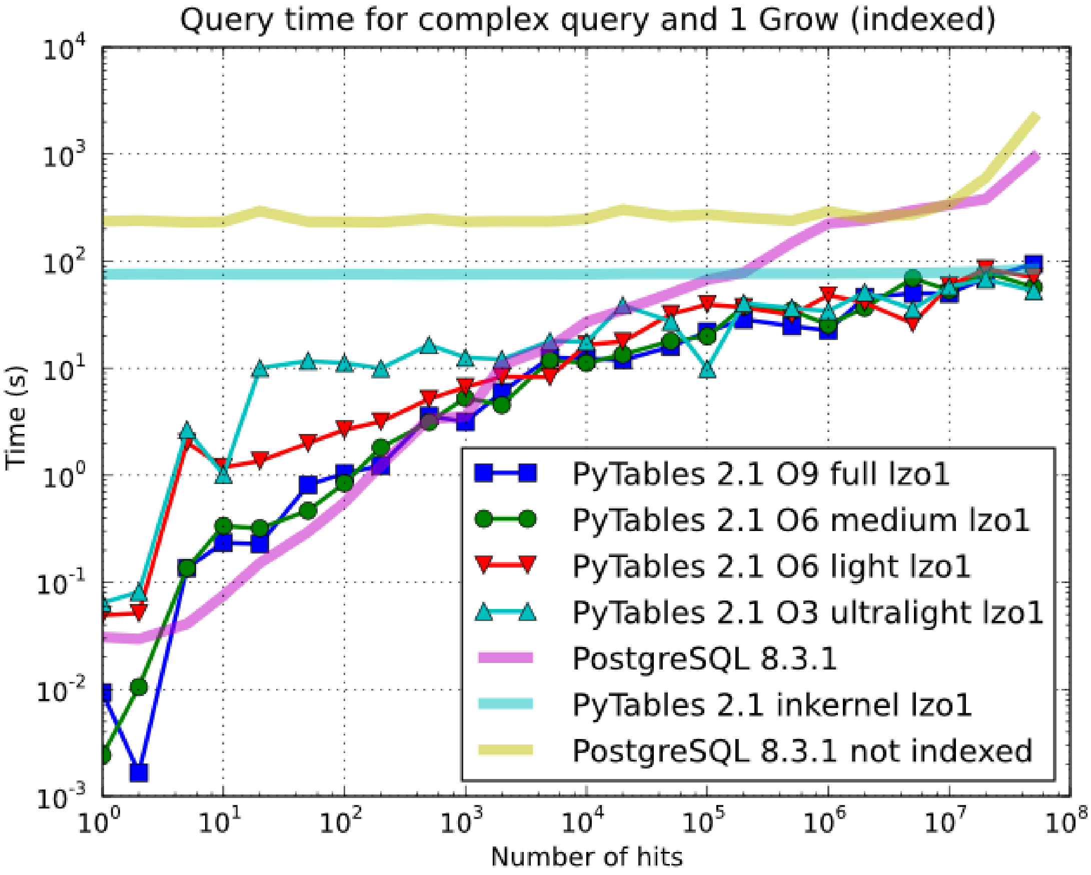


Fig. 5.9: **Figure 9. Times for complex queries with a cold cache (mean of 5 first random queries) for different optimization levels. Benchmark made on a machine with Intel Core2 (64-bit) @ 3 GHz processor with RAID-0 disk storage.**

Of course, compression also has an effect when doing indexed queries, although not very noticeable, as can be seen in *Figure 10*. As you can see, the difference between using no compression and using Zlib or LZO is very little, although LZO achieves relatively better performance generally speaking.

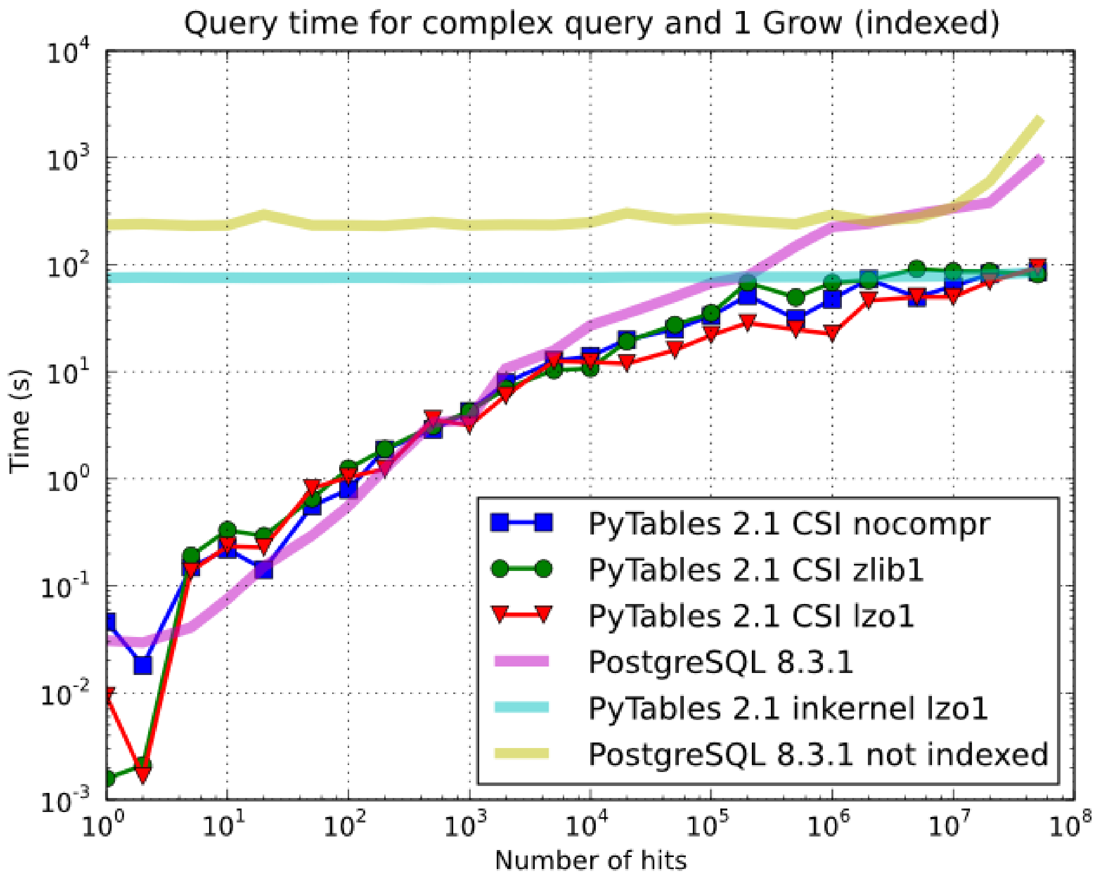


Fig. 5.10: **Figure 10. Times for complex queries with a cold cache (mean of 5 first random queries) for different compressors.**

You can find a more complete description and benchmarks about OPSI, the indexing system of PyTables (>= 2.3) in *[OPSI]*.

5.2.3 Indexing and Solid State Disks (SSD)

Lately, the long promised Solid State Disks (SSD for brevity) with decent capacities and affordable prices have finally hit the market and will probably stay in coexistence with the traditional spinning disks for the foreseeable future (separately or forming *hybrid* systems). SSD have many advantages over spinning disks, like much less power consumption and better throughput. But of paramount importance, specially in the context of accelerating indexed queries, is its very reduced latency during disk seeks, which is typically 100x better than traditional disks. Such a huge improvement has to have a clear impact in reducing the query times, specially when the selectivity is high (i.e. the number of hits is small).

In order to offer an estimate on the performance improvement we can expect when using a low-latency SSD instead of traditional spinning disks, the benchmark in the previous section has been repeated, but this time using a single SSD disk instead of the four spinning disks in RAID-0. The result can be seen in *Figure 11*. There one can see how a query in a table of 1 billion of rows with 100 hits took just 1 tenth of second when using a SSD, instead of 1 second that needed the RAID made of spinning disks. This factor of 10x of speed-up for high-selectivity queries is nothing to sneeze at, and should be kept in mind when really high performance in queries is needed. It is also interesting that using compression with LZO does have a clear advantage over when no compression is done.

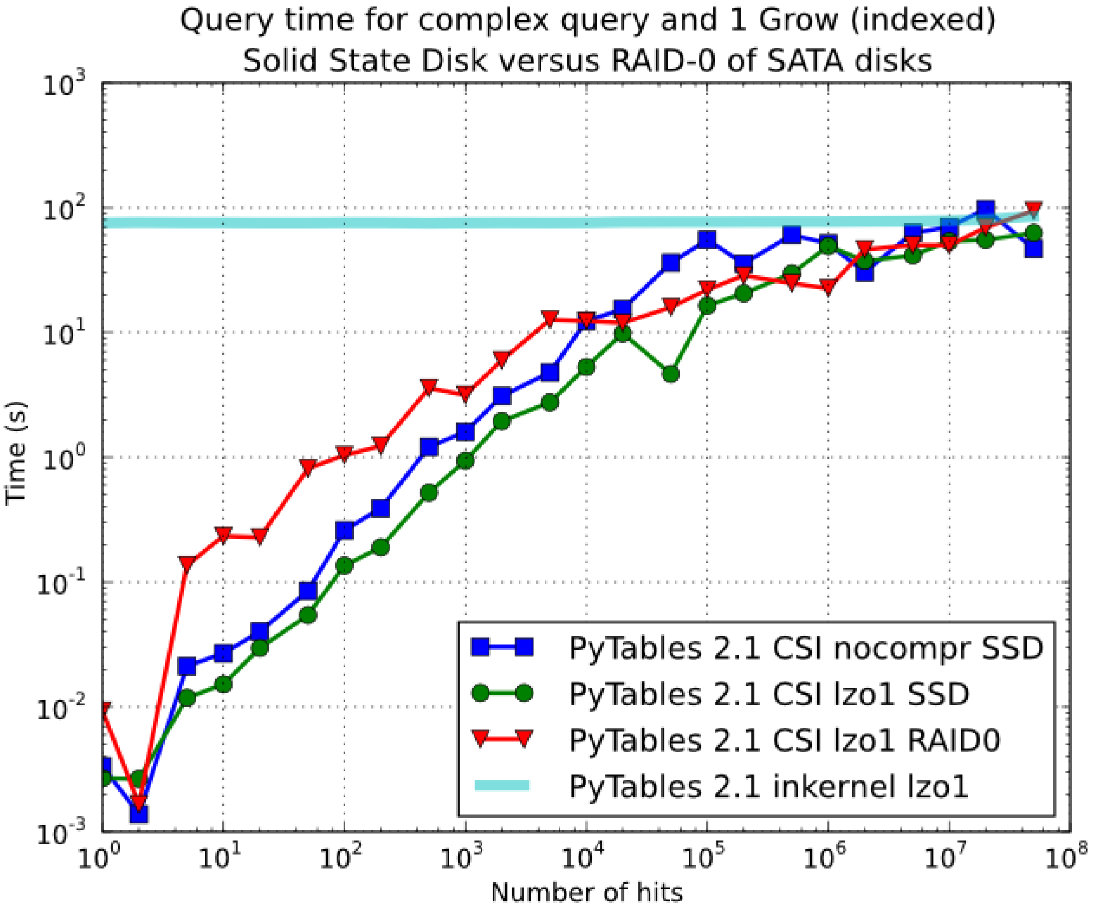


Fig. 5.11: **Figure 11. Times for complex queries with a cold cache (mean of 5 first random queries) for different disk storage (SSD vs spinning disks).**

Finally, we should remark that SSD can't compete with traditional spinning disks in terms of capacity as they can only provide, for a similar cost, between 1/10th and 1/50th of the size of traditional disks. It is here where the compression capabilities of PyTables can be very helpful because both tables and indexes can be compressed and the final space can be reduced by typically 2x to 5x (4x to 10x when compared with traditional relational databases). Best of all, as already mentioned, performance is not degraded when compression is used, but actually *improved*. So, by using PyTables and SSD you can query larger datasets that otherwise would require spinning disks when using other databases

In fact, we were unable to run the PostgreSQL benchmark in this case because the space needed exceeded the capacity of our SSD., while allowing improvements in the speed of indexed queries between 2x (for medium to low selectivity queries) and 10x (for high selectivity queries).

5.2.4 Achieving ultimate speed: sorted tables and beyond

Warning: Sorting a large table is a costly operation. The next procedure should only be performed when your dataset is mainly read-only and meant to be queried many times.

When querying large tables, most of the query time is spent in locating the interesting rows to be read from disk. In some occasions, you may have queries whose result depends *mainly* of one single column (a query with only one single condition is the trivial example), so we can guess that sorting the table by this column would lead to locate the interesting rows in a much more efficient way (because they would be mostly *contiguous*). We are going to confirm this guess.

For the case of the query that we have been using in the previous sections:

```
result = [row['col2'] for row in table.where(
            '''(((col4 >= lim1) & (col4 < lim2)) |
               ((col2 > lim3) & (col2 < lim4)) &
               ((col1+3.1*col2+col3*col4) > lim5))''')]
```

it is possible to determine, by analysing the data distribution and the query limits, that col4 is such a *main column*. So, by ordering the table by the col4 column (for example, by specifying setting the column to sort by in the sortby parameter in the `Table.copy()` method and re-indexing col2 and col4 afterwards, we should get much faster performance for our query. This is effectively demonstrated in *Figure 12*, where one can see how queries with a low to medium (up to 10000) number of hits can be done in around 1 tenth of second for a RAID-0 setup and in around 1 hundredth of second for a SSD disk. This represents up to more that 100x improvement in speed with respect to the times with unsorted tables. On the other hand, when the number of hits is large (> 1 million), the query times grow almost linearly, showing a near-perfect scalability for both RAID-0 and SSD setups (the sequential access to disk becomes the bottleneck in this case).

Even though we have shown many ways to improve query times that should fulfill the needs of most of people, for those needing more, you can for sure discover new optimization opportunities. For example, querying against sorted tables is limited mainly by sequential access to data on disk and data compression capability, so you may want to read *Fine-tuning the chunksize*, for ways on improving this aspect. Reading the other sections of this chapter will help in finding new roads for increasing the performance as well. You know, the limit for stopping the optimization process is basically your imagination (but, most plausibly, your available time ;-).

5.3 Compression issues

One of the beauties of PyTables is that it supports compression on tables and arrays [2], although it is not used by default. Compression of big amounts of data might be a bit controversial feature, because it has a legend of being a very big consumer of CPU time resources. However, if you are willing to check if compression can help not only by reducing your dataset file size but *also* by improving I/O efficiency, specially when dealing with very large datasets, keep reading.

5.3.1 A study on supported compression libraries

The compression library used by default is the *Zlib* (see *[ZLIB]*). Since HDF5 *requires* it, you can safely use it and expect that your HDF5 files will be readable on any other platform that has HDF5 libraries installed. Zlib provides 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.

[2] Except for Array objects.

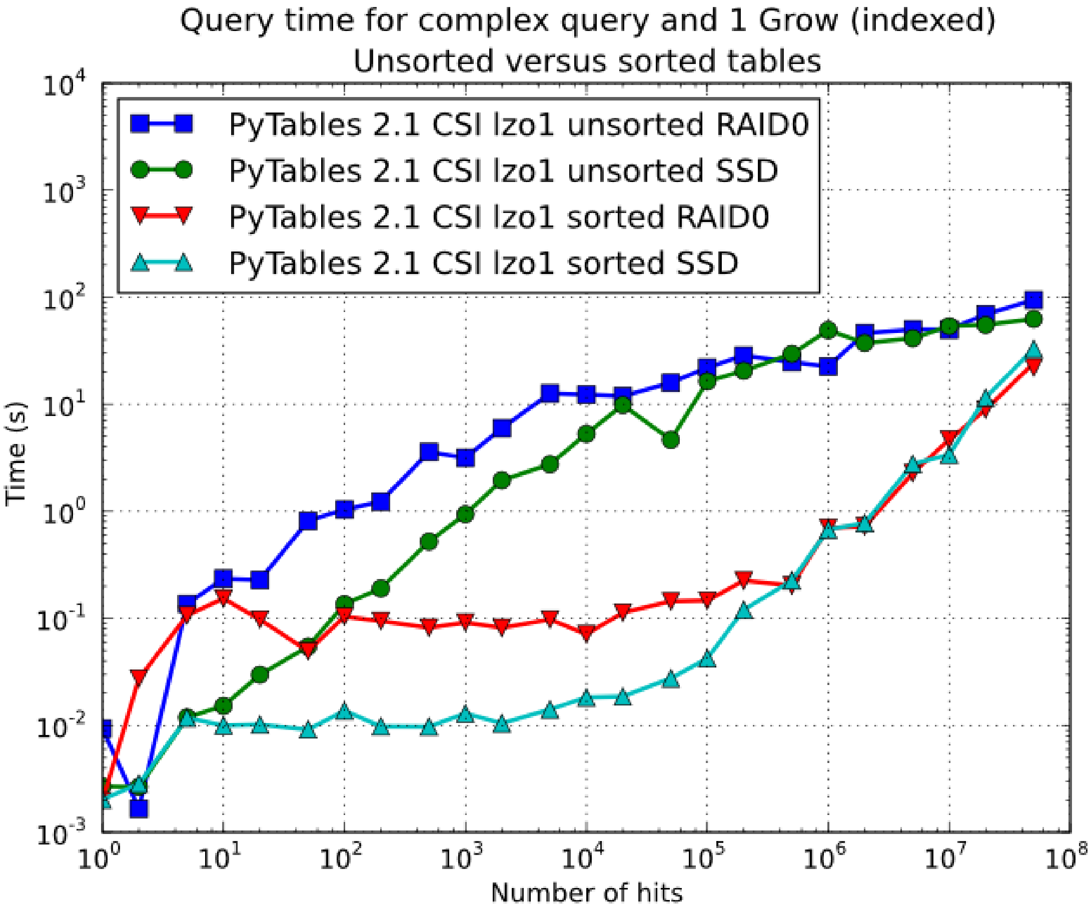


Fig. 5.12: **Figure 12. Times for complex queries with a cold cache (mean of 5 first random queries) for unsorted and sorted tables.**

However, in some situations it is critical to have a *very good decompression speed* (at the expense of lower compression ratios or more CPU wasted on compression, as we will see soon). In others, the emphasis is put in achieving the *maximum compression ratios*, no matter which reading speed will result. This is why support for two additional compressors has been added to PyTables: LZO (see *[LZO]*) and bzip2 (see *[BZIP2]*). Following the author of LZO (and checked by the author of this section, as you will see soon), LZO offers pretty fast compression and extremely fast decompression. In fact, LZO is so fast when compressing/decompressing that it may well happen (that depends on your data, of course) that writing or reading a compressed dataset is sometimes faster than if it is not compressed at all (specially when dealing with extremely large datasets). This fact is very important, specially if you have to deal with very large amounts of data. Regarding bzip2, it has a reputation of achieving excellent compression ratios, but at the price of spending much more CPU time, which results in very low compression/decompression speeds.

Be aware that the LZO and bzip2 support in PyTables is not standard on HDF5, so if you are going to use your PyTables files in other contexts different from PyTables you will not be able to read them. Still, see the *ptrepack* (where the ptrepack utility is described) to find a way to free your files from LZO or bzip2 dependencies, so that you can use these compressors locally with the warranty that you can replace them with Zlib (or even remove compression completely) if you want to use these files with other HDF5 tools or platforms afterwards.

In order to allow you to grasp what amount of compression can be achieved, and how this affects performance, a series of experiments has been carried out. All the results presented in this section (and in the next one) have been obtained with synthetic data and using PyTables 1.3. Also, the tests have been conducted on a IBM OpenPower 720 (e-series) with a PowerPC G5 at 1.65 GHz and a hard disk spinning at 15K RPM. As your data and platform may be totally different for your case, take this just as a guide because your mileage may vary. Finally, and to be able to play with tables with a number of rows as large as possible, the record size has been chosen to be small (16 bytes). Here is its definition:

```
class Bench(IsDescription):
    var1 = StringCol(length=4)
    var2 = IntCol()
    var3 = FloatCol()
```

With this setup, you can look at the compression ratios that can be achieved in *Figure 13*. As you can see, LZO is the compressor that performs worse in this sense, but, curiously enough, there is not much difference between Zlib and bzip2.

Also, PyTables lets you select different compression levels for Zlib and bzip2, although you may get a bit disappointed by the small improvement that these compressors show when dealing with a combination of numbers and strings as in our example. As a reference, see plot *Figure 14* for a comparison of the compression achieved by selecting different levels of Zlib. Very oddly, the best compression ratio corresponds to level 1 (!). See later for an explanation and more figures on this subject.

Have also a look at *Figure 15*. It shows how the speed of writing rows evolves as the size (number of rows) of the table grows. Even though in these graphs the size of one single row is 16 bytes, you can most probably extrapolate these figures to other row sizes.

In *Figure 16* you can see how compression affects the reading performance. In fact, what you see in the plot is an *in-kernel selection* speed, but provided that this operation is very fast (see *In-kernel searches*), we can accept it as an actual read test. Compared with the reference line without compression, the general trend here is that LZO does not affect too much the reading performance (and in some points it is actually better), Zlib makes speed drop to a half, while bzip2 is performing very slow (up to 8x slower).

Also, in the same *Figure 16* you can notice some strange peaks in the speed that we might be tempted to attribute to libraries on which PyTables relies (HDF5, compressors...), or to PyTables itself. However, *Figure 17* reveals that, if we put the file in the filesystem cache (by reading it several times before, for example), the evolution of the performance is much smoother. So, the most probable explanation would be that such peaks are a consequence of the underlying OS filesystem, rather than a flaw in PyTables (or any other library behind it). Another consequence that can be derived from the aforementioned plot is that LZO decompression performance is much better than Zlib, allowing an improvement in overall speed of more than 2x, and perhaps more important, the read performance for really large datasets (i.e. when they do not fit in the OS filesystem cache) can be actually *better* than not using compression at all.

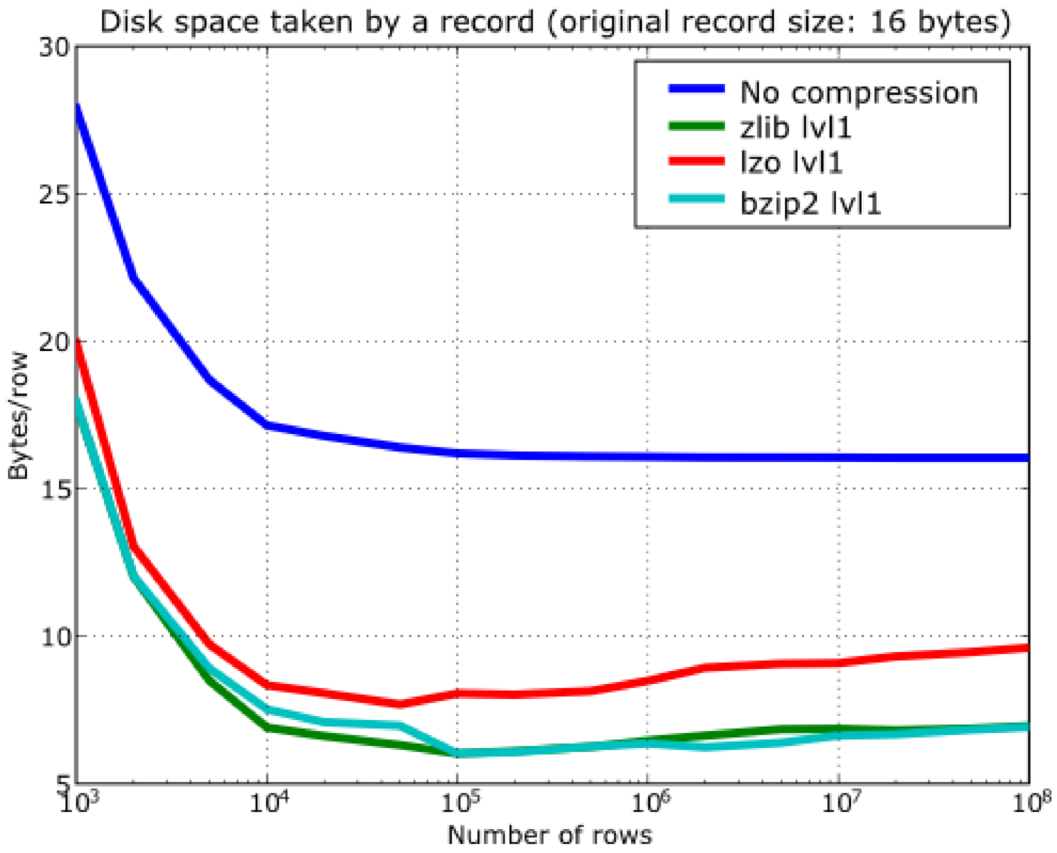


Fig. 5.13: **Figure 13. Comparison between different compression libraries.**

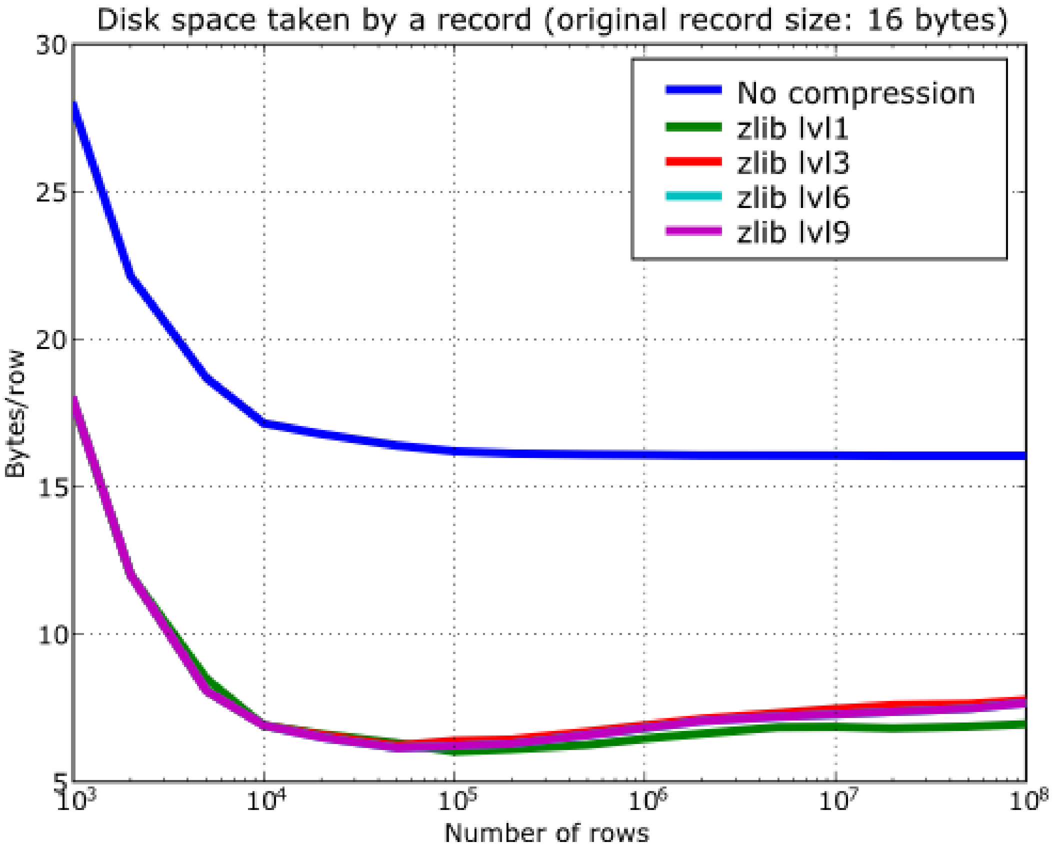


Fig. 5.14: **Figure 14. Comparison between different compression levels of Zlib.**

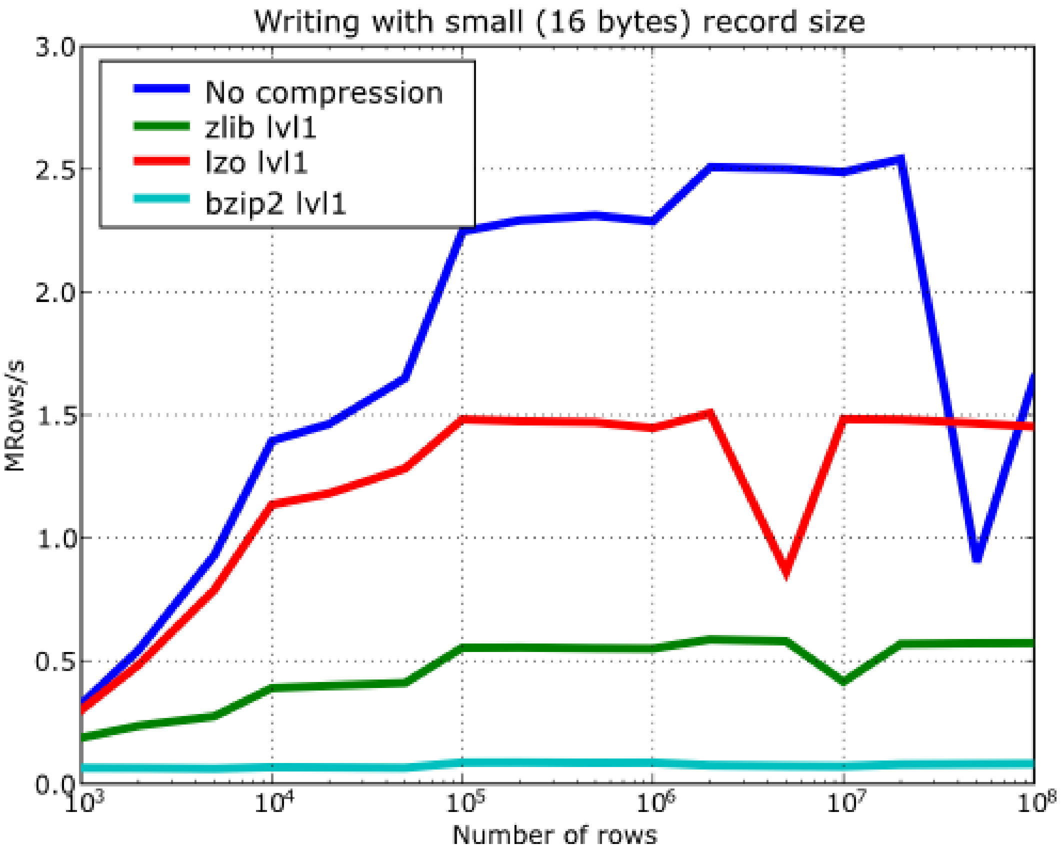


Fig. 5.15: **Figure 15. Writing tables with several compressors.**

Finally, one can see that reading performance is very badly affected when bzip2 is used (it is 10x slower than LZO and 4x than Zlib), but this was somewhat expected anyway.

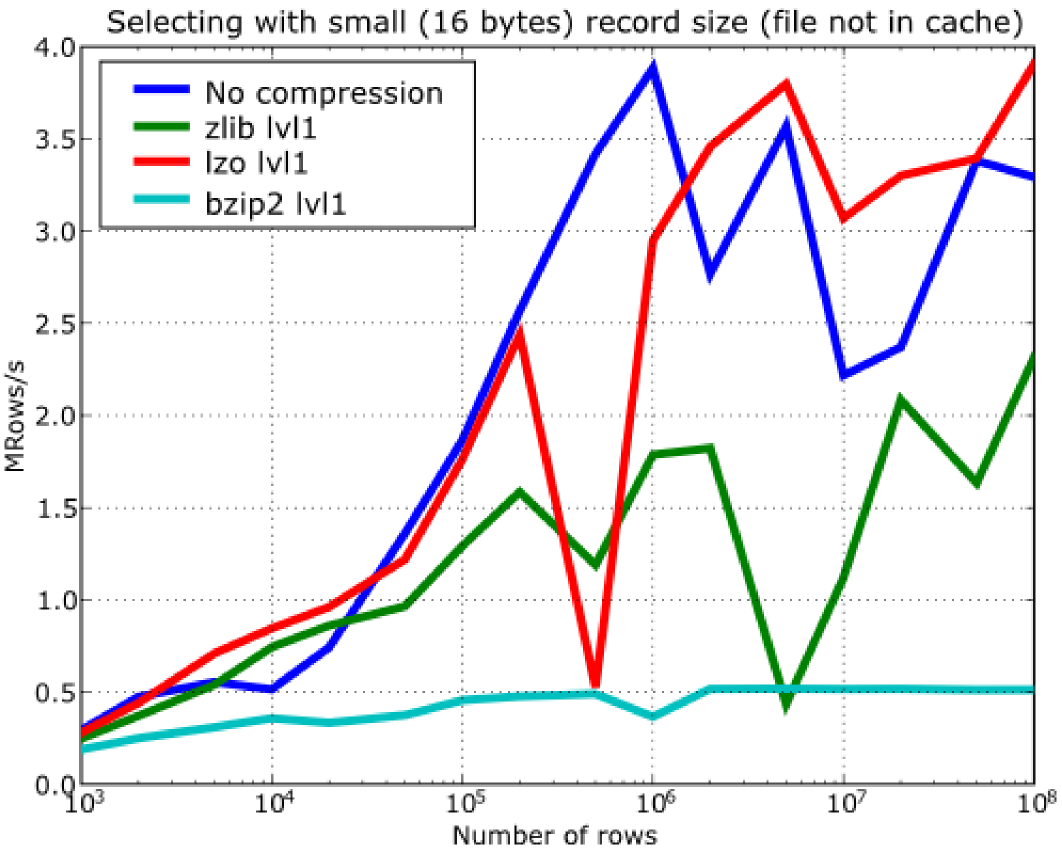


Fig. 5.16: **Figure 16. Selecting values in tables with several compressors. The file is not in the OS cache.**

So, generally speaking and looking at the experiments above, you can expect that LZO will be the fastest in both compressing and decompressing, but the one that achieves the worse compression ratio (although that may be just OK for many situations, specially when used with shuffling - see *Shuffling (or how to make the compression process more effective)*). bzip2 is the slowest, by large, in both compressing and decompressing, and besides, it does not achieve any better compression ratio than Zlib. Zlib represents a balance between them: it's somewhat slow compressing (2x) and decompressing (3x) than LZO, but it normally achieves better compression ratios.

Finally, by looking at the plots *Figure 18*, *Figure 19*, and the aforementioned *Figure 14* you can see why the recommended compression level to use for all compression libraries is 1. This is the lowest level of compression, but as the size of the underlying HDF5 chunk size is normally rather small compared with the size of compression buffers, there is not much point in increasing the latter (i.e. increasing the compression level). Nonetheless, in some situations (like for example, in extremely large tables or arrays, where the computed chunk size can be rather large) you may want to check, on your own, how the different compression levels do actually affect your application.

You can select the compression library and level by setting the complib and complevel keywords in the Filters class (see *The Filters class*). A compression level of 0 will completely disable compression (the default), 1 is the less memory and CPU time demanding level, while 9 is the maximum level and the most memory demanding and CPU 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.

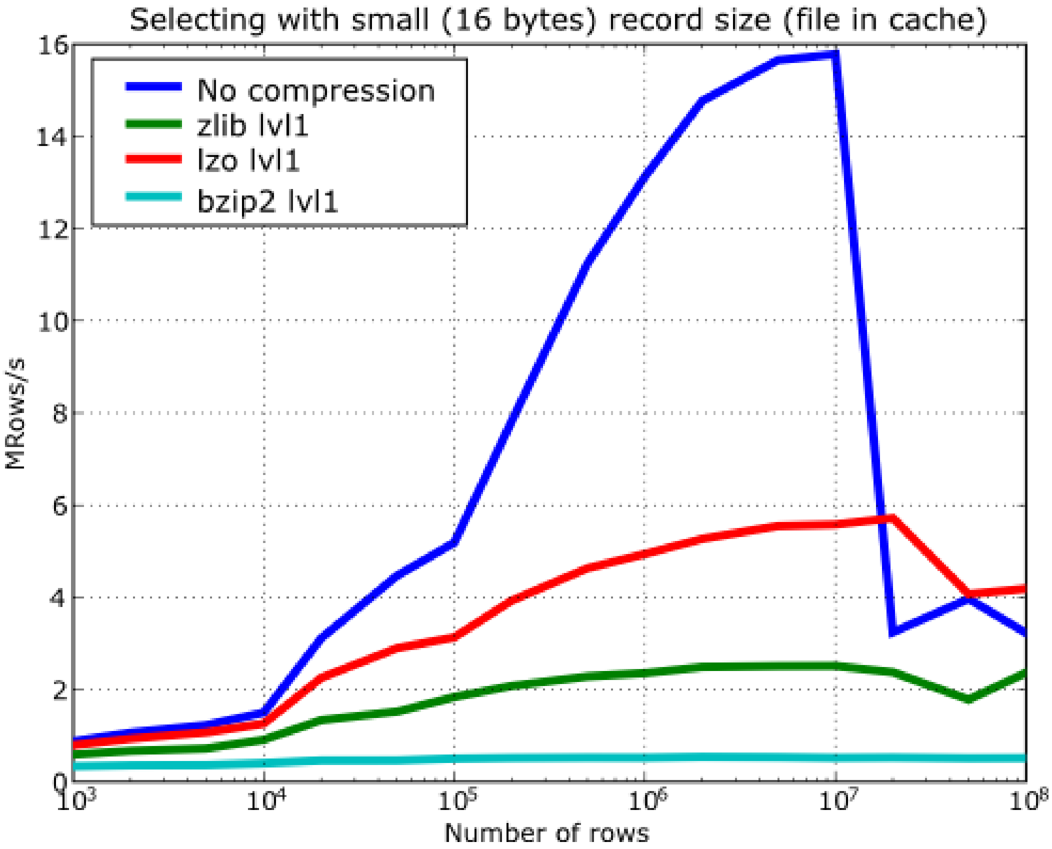


Fig. 5.17: **Figure 17. Selecting values in tables with several compressors. The file is in the OS cache.**

So, in conclusion, if your ultimate goal is writing and reading as fast as possible, choose LZO. If you want to reduce as much as possible your data, while retaining acceptable read speed, choose Zlib. Finally, if portability is important for you, Zlib is your best bet. So, when you want to use bzip2? Well, looking at the results, it is difficult to recommend its use in general, but you may want to experiment with it in those cases where you know that it is well suited for your data pattern (for example, for dealing with repetitive string datasets).

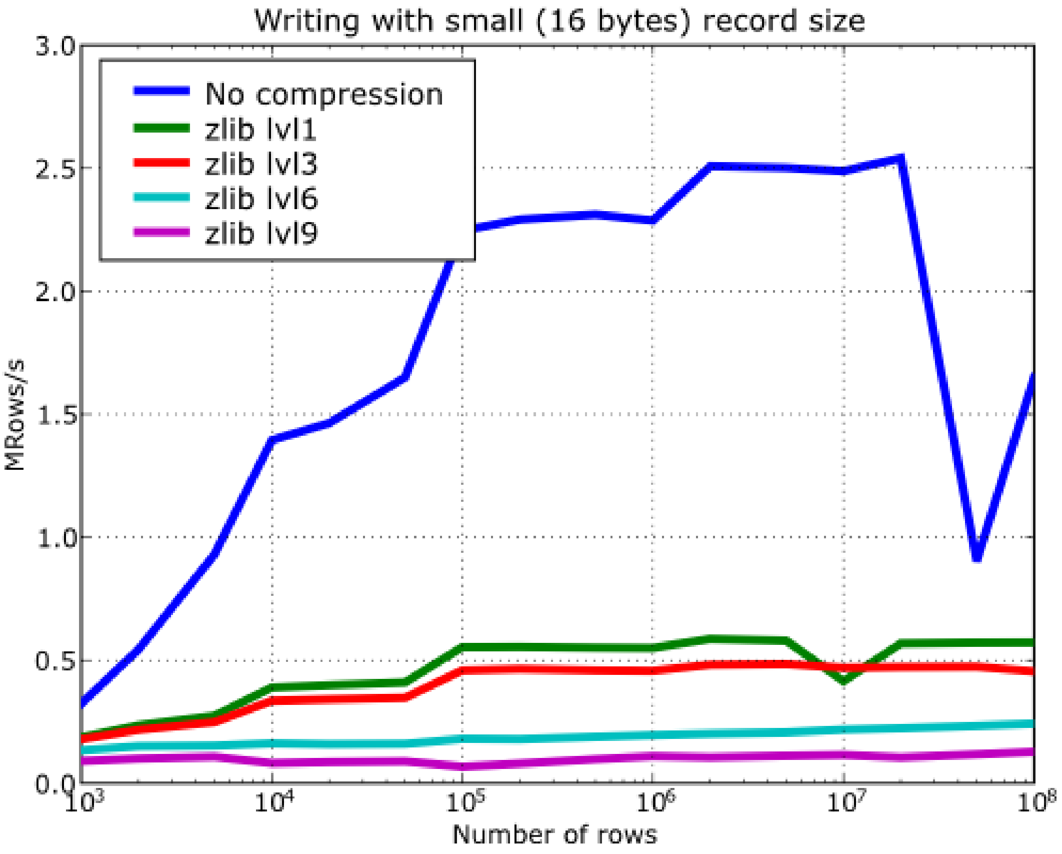


Fig. 5.18: **Figure 18. Writing in tables with different levels of compression.**

5.3.2 Shuffling (or how to make the compression process more effective)

The HDF5 library provides an interesting filter that can leverage the results of your favorite compressor. Its name is *shuffle*, and because it can greatly benefit compression and it does not take many CPU resources (see below for a justification), it is active *by default* in PyTables whenever compression is activated (independently of the chosen compressor). It is deactivated when compression is off (which is the default, as you already should know). Of course, you can deactivate it if you want, but this is not recommended.

So, how does this mysterious filter exactly work? From the HDF5 reference manual:

```
"The shuffle filter de-interlaces a block of data by reordering the
bytes. All the bytes from one consistent byte position of each data
element are placed together in one block; all 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
```

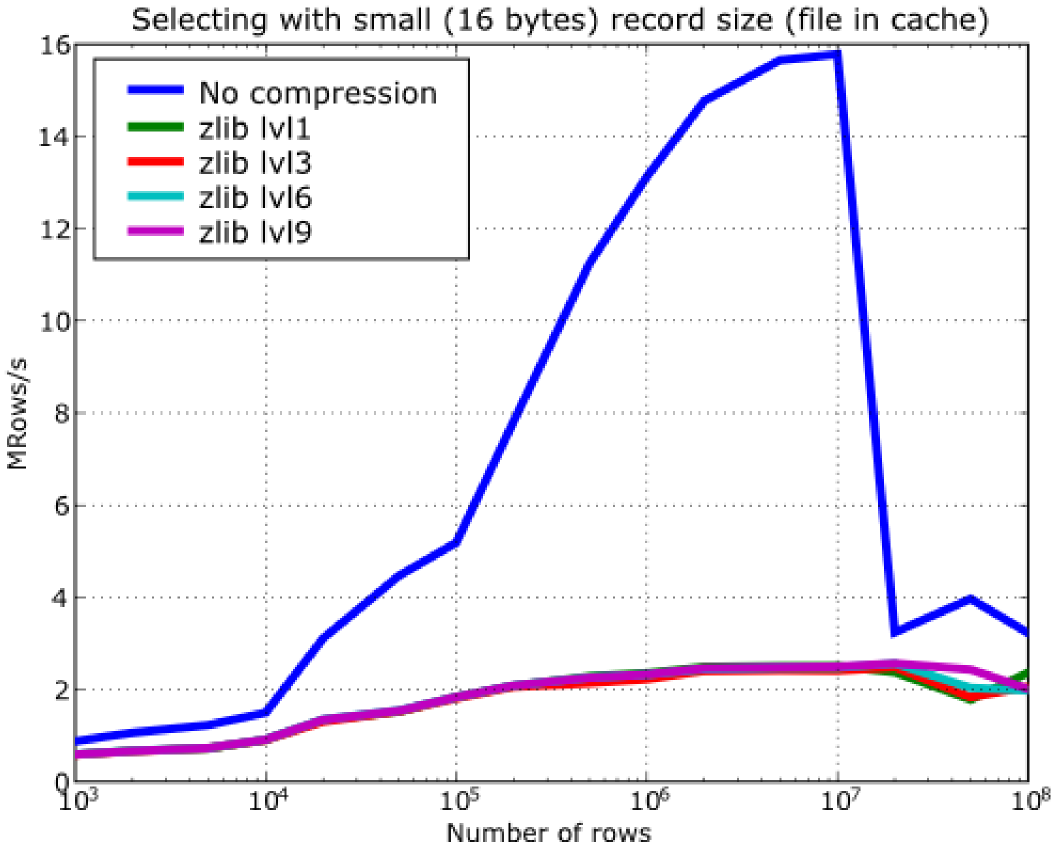


Fig. 5.19: **Figure 19. Selecting values in tables with different levels of compression. The file is in the OS cache.**

```
datatype stored as 012301230123, shuffling will re-order data as
000111222333. This can be a valuable step in an effective compression
algorithm because the bytes in each byte position are often closely
related to each other and putting them together can increase the
compression ratio."
```

In *Figure 20* you can see a benchmark that shows how the *shuffle* filter can help the different libraries in compressing data. In this experiment, shuffle has made LZO compress almost 3x more (!), while Zlib and bzip2 are seeing improvements of 2x. Once again, the data for this experiment is synthetic, and *shuffle* seems to do a great work with it, but in general, the results will vary in each case [3].

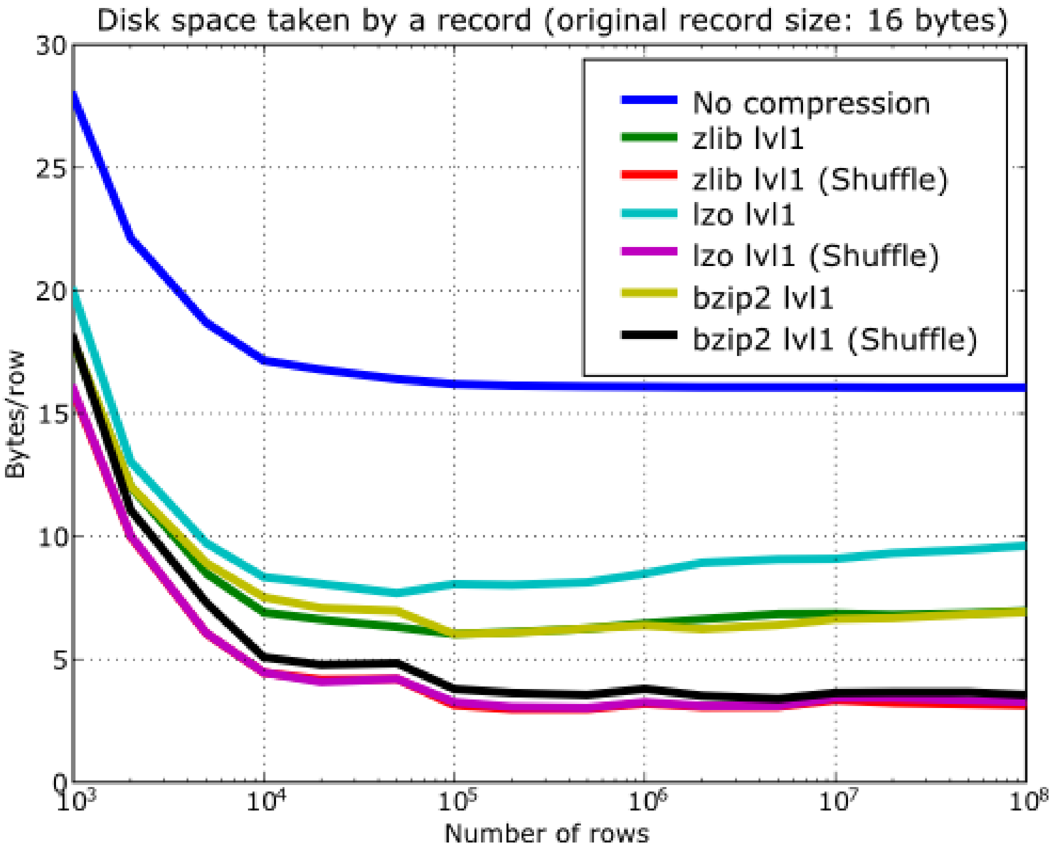


Fig. 5.20: **Figure 20. Comparison between different compression libraries with and without the shuffle filter.**

At any rate, the most remarkable fact about the *shuffle* filter is the relatively high level of compression that compressor filters can achieve when used in combination with it. A curious thing to note is that the Bzip2 compression rate does not seem very much improved (less than a 40%), and what is more striking, Bzip2+shuffle does compress quite *less* than Zlib+shuffle or LZO+shuffle combinations, which is kind of unexpected. The thing that seems clear is that Bzip2 is not very good at compressing patterns that result of shuffle application. As always, you may want to experiment with your own data before widely applying the Bzip2+shuffle combination in order to avoid surprises.

Now, how does shuffling affect performance? Well, if you look at plots *Figure 21*, *Figure 22* and *Figure 23*, you will get a somewhat unexpected (but pleasant) surprise. Roughly, *shuffle* makes the writing process (shuffling+compressing) faster (approximately a 15% for LZO, 30% for Bzip2 and a 80% for Zlib), which is an interesting result by itself. But

[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.

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).

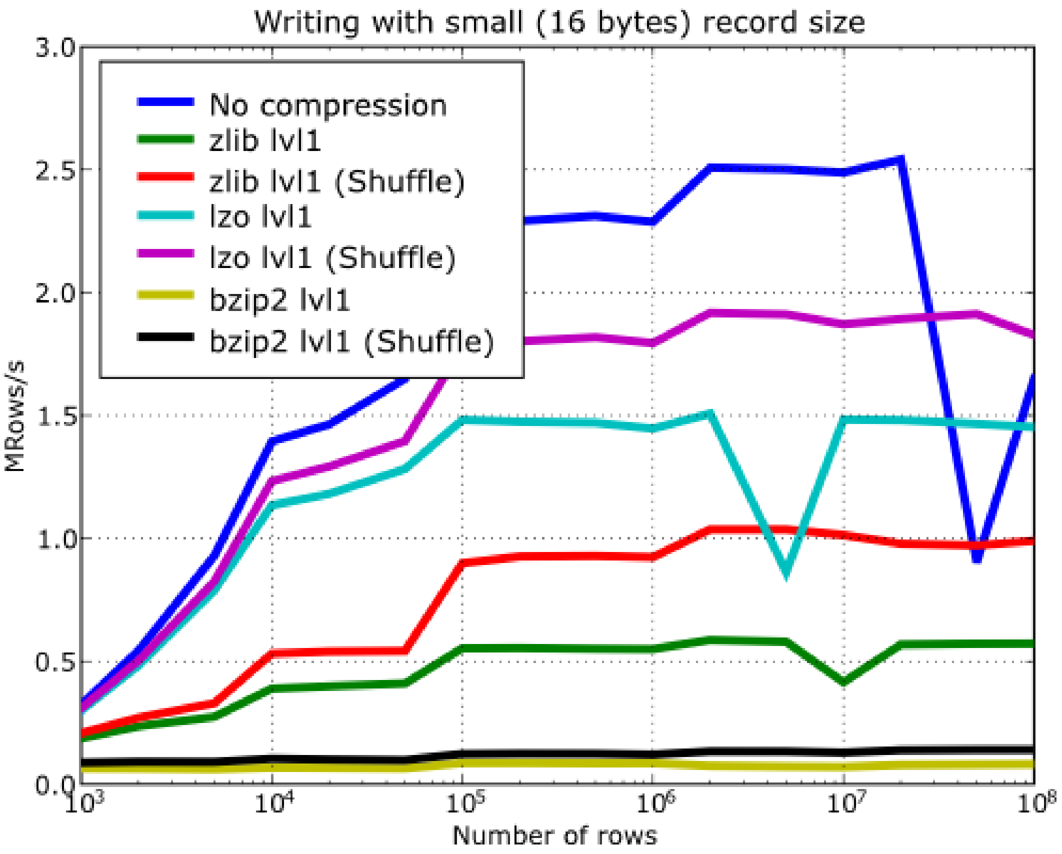


Fig. 5.21: **Figure 21. Writing with different compression libraries with and without the shuffle filter.**

You may wonder why introducing another filter in the write/read pipelines does effectively accelerate the throughput. Well, maybe data elements are more similar or related column-wise than row-wise, i.e. contiguous elements in the same column are more alike, so shuffling makes the job of the compressor easier (faster) and more effective (greater ratios). As a side effect, compressed chunks do fit better in the CPU cache (at least, the chunks are smaller!) so that the process of unshuffle/decompress can make a better use of the cache (i.e. reducing the number of CPU cache faults).

So, given the potential gains (faster writing and reading, but specially much improved compression level), it is a good thing to have such a filter enabled by default in the battle for discovering redundancy when you want to compress your data, just as PyTables does.

5.4 Using Psyco

Psyco (see *[PSYCO]*) is a kind of specialized compiler for Python that typically accelerates Python applications with no change in source code. You can think of Psyco as a kind of just-in-time (JIT) compiler, a little bit like Java's, that emits machine code on the fly instead of interpreting your Python program step by step. The result is that your unmodified Python programs run faster.

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

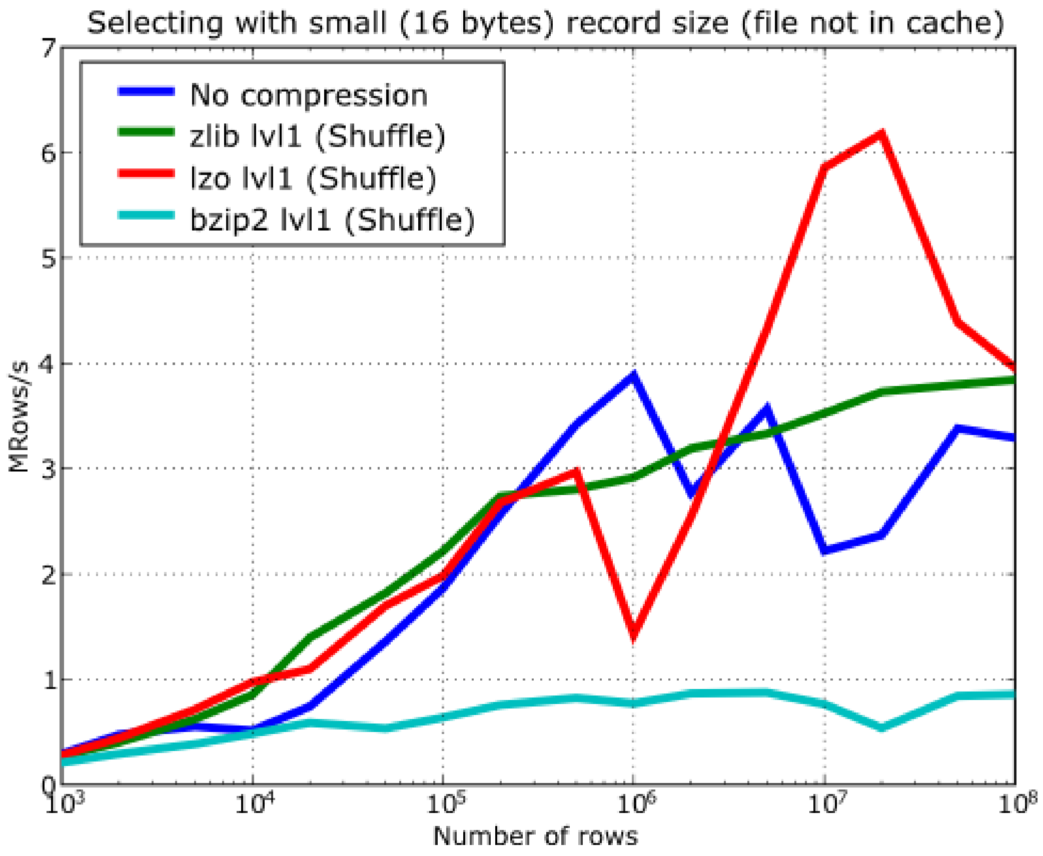


Fig. 5.22: **Figure 22. Reading with different compression libraries with the shuffle filter. The file is not in OS cache.**

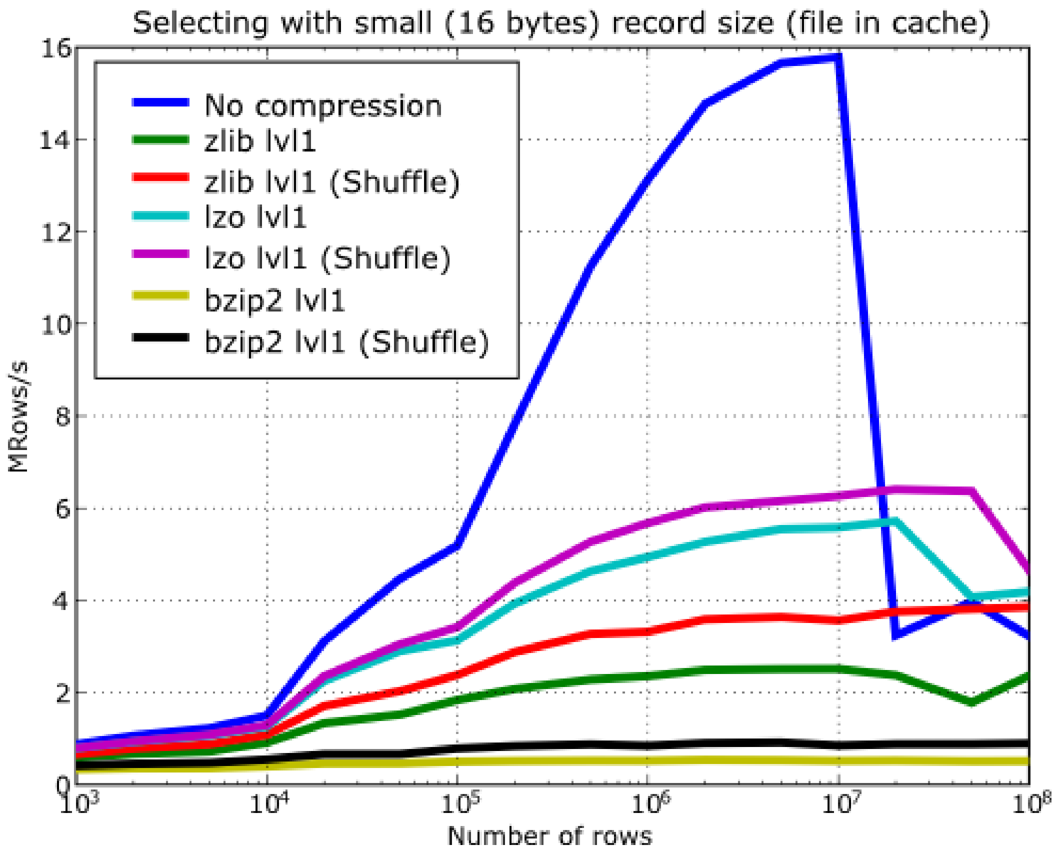


Fig. 5.23: **Figure 23. Reading with different compression libraries with and without the shuffle filter. The file is in OS cache.**

386 architectures, so if you are using other architectures, you are out of luck (and, moreover, it seems that there are no plans to support other platforms). Besides, with the addition of flexible (and very fast) in-kernel queries (by the way, they cannot be optimized at all by Psyco), the use of Psyco will only help in rather few scenarios. In fact, the only important situation that you might benefit right now from using Psyco (I mean, in PyTables contexts) is for speeding-up the write speed in tables when using the Row interface (see *The Row class*). But again, this latter case can also be accelerated by using the `Table.append()` method and building your own buffers [4].

As an example, imagine that you have a small script that reads and selects data over a series of datasets, like this:

```
def read_file(filename):
    "Select data from all the tables in filename"
    fileh = open_file(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(read_file("myfile.h5"))
```

In order to accelerate this piece of code, you can rewrite your main program to look like:

```
if __name__=="__main__":
    import psyco
    psyco.bind(read_file)
    print(read_file("myfile.h5"))
```

That's all! From now on, each time that you execute your Python script, Psyco will deploy its sophisticated algorithms so as to accelerate your calculations.

You can see in the graphs *Figure 24* and *Figure 25* how much I/O speed improvement you can get by using Psyco. By looking at this figures you can get an idea if these improvements are of your interest or not. In general, if you are not going to use compression you will take advantage of Psyco if your tables are medium sized (from a thousand to a million rows), and this advantage will disappear progressively when the number of rows grows well over one million. However if you use compression, you will probably see improvements even beyond this limit (see *Compression issues*). As always, there is no substitute for experimentation with your own dataset.

5.5 Getting the most from the node LRU cache

One limitation of the initial versions of PyTables was that they needed to load all nodes in a file completely before being ready to deal with them, making the opening times for files with a lot of nodes very high and unacceptable in many cases.

Starting from PyTables 1.2 on, a new lazy node loading schema was setup that avoids loading all the nodes of the *object tree* in memory. In addition, a new LRU cache was introduced in order to accelerate the access to already visited nodes. This cache (one per file) is responsible for keeping up the most recently visited nodes in memory and discard the least recent used ones. This represents a big advantage over the old schema, not only in terms of memory usage (as there is no need to load *every* node in memory), but it also adds very convenient optimizations for working interactively like, for example, speeding-up the opening times of files with lots of nodes, allowing to open almost any kind of file in typically less than one tenth of second (compare this with the more than 10 seconds for files with more than 10000 nodes in PyTables pre-1.2 era) as well as optimizing the access to frequently visited nodes. See for more info on the advantages (and also drawbacks) of this approach.

[4] So, there is not much point in using Psyco with recent versions of PyTables anymore.

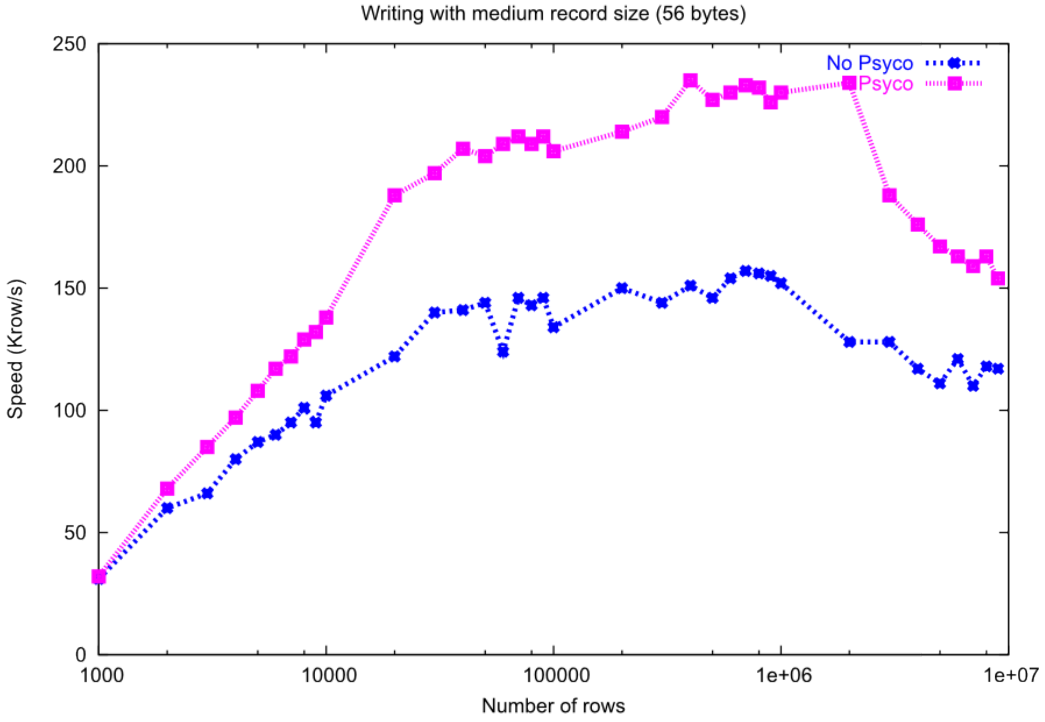


Fig. 5.24: **Figure 24. Writing tables with/without Psyco.**

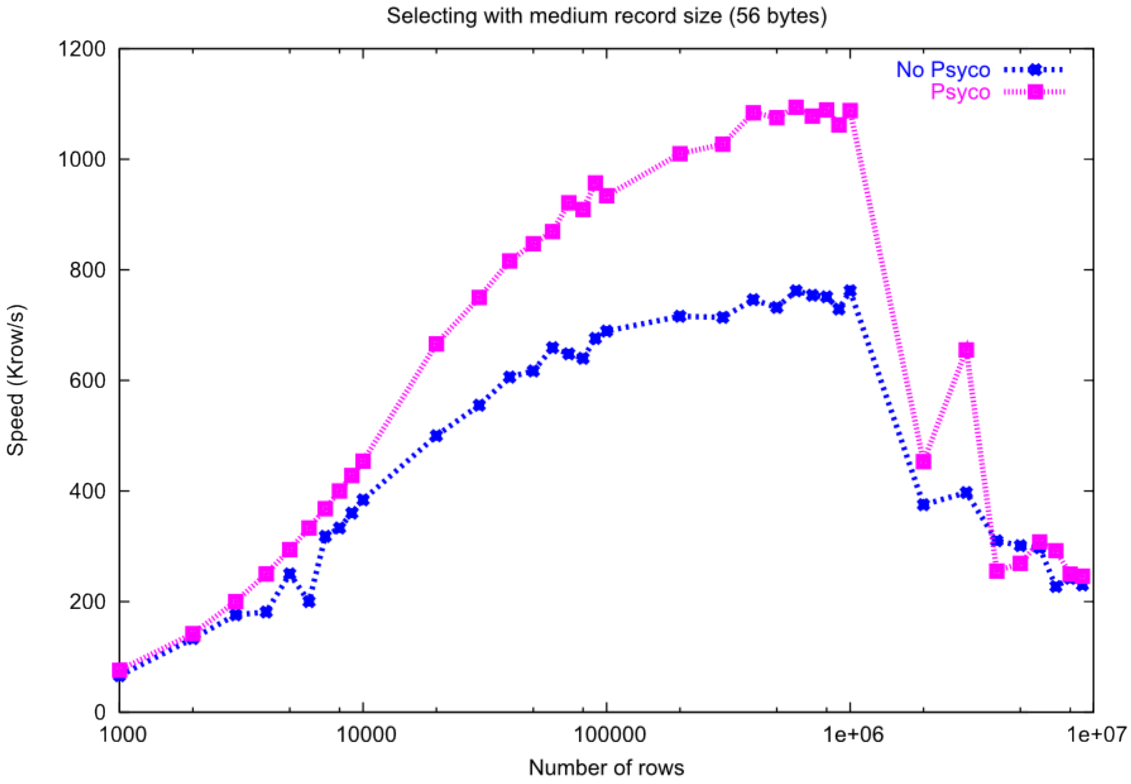


Fig. 5.25: **Figure 25. Reading tables with/without Psyco.**

One thing that deserves some discussion is the election of the parameter that sets the maximum amount of nodes to be kept in memory at any time. As PyTables is meant to be deployed in machines that can have potentially low memory, the default for it is quite conservative (you can look at its actual value in the *parameters.NODE_CACHE_SLOTS* parameter in module `tables/parameters.py`). However, if you usually need to deal with files that have many more nodes than the maximum default, and you have a lot of free memory in your system, then you may want to experiment in order to see which is the appropriate value of *parameters.NODE_CACHE_SLOTS* that fits better your needs.

As an example, look at the next code:

```
def browse_tables(filename):
    fileh = open_file(filename,'a')
    group = fileh.root.newgroup
    for j in range(10):
        for tt in fileh.walk_nodes(group, "Table"):
            title = tt.attrs.TITLE
            for row in tt:
                pass
    fileh.close()
```

We will be running the code above against a couple of files having a `/newgroup` containing 100 tables and 1000 tables respectively. In addition, this benchmark is run twice for two different values of the LRU cache size, specifically 256 and 1024. You can see the results in *table*.

Table 5.1: Retrieval speed and memory consumption depending on the number of nodes in LRU cache.

Number:		**100 nodes**				**1000 nodes**			
Mem and Speed		**Memory (MB)**		**Time (ms)**		**Memory (MB)**		**Time (ms)**	
Node is coming from...	**Cache size**	**256**	**1024**	**256**	**1024**	**256**	**1024**	**256**	**1024**
Disk		14	14	1.24	1.24	51	66	1.33	1.31
Cache		14	14	0.53	0.52	65	73	1.35	0.68

From the data in *table*, one can see that when the number of objects that you are dealing with does fit in cache, you will get better access times to them. Also, incrementing the node cache size effectively consumes more memory *only* if the total nodes exceeds the slots in cache; otherwise the memory consumption remains the same. It is also worth noting that incrementing the node cache size in the case you want to fit all your nodes in cache does not take much more memory than being too conservative. On the other hand, it might happen that the speed-up that you can achieve by allocating more slots in your cache is not worth the amount of memory used.

Also worth noting is that if you have a lot of memory available and performance is absolutely critical, you may want to try out a negative value for *parameters.NODE_CACHE_SLOTS*. This will cause that all the touched nodes will be kept in an internal dictionary and this is the faster way to load/retrieve nodes. However, and in order to avoid a large memory consumption, the user will be warned when the number of loaded nodes will reach the `-NODE_CACHE_SLOTS` value.

Finally, a value of zero in *parameters.NODE_CACHE_SLOTS* means that any cache mechanism is disabled.

At any rate, if you feel that this issue is important for you, there is no replacement for setting your own experiments up in order to proceed to fine-tune the *parameters.NODE_CACHE_SLOTS* parameter.

Note: PyTables >= 2.3 sports an optimized LRU cache node written in C, so you should expect significantly faster LRU cache operations when working with it.

Note: Numerical results reported in *table* have been obtained with PyTables < 3.1. In PyTables 3.1 the node cache mechanism has been completely redesigned so while all comments above are still valid, numerical values could be a little bit different from the ones reported in *table*.

5.6 Compacting your PyTables files

Let's suppose that you have a file where you have made a lot of row deletions on one or more tables, or deleted many leaves or even entire subtrees. These operations might leave *holes* (i.e. space that is not used anymore) in your files that may potentially affect not only the size of the files but, more importantly, the performance of I/O. This is because when you delete a lot of rows in a table, the space is not automatically recovered on the fly. In addition, if you add many more rows to a table than specified in the expectedrows keyword at creation time this may affect performance as well, as explained in *Informing PyTables about expected number of rows in tables or arrays*.

In order to cope with these issues, you should be aware that PyTables includes a handy utility called ptrepack which can be very useful not only to compact *fragmented* files, but also to adjust some internal parameters in order to use better buffer and chunk sizes for optimum I/O speed. Please check the *ptrepack* for a brief tutorial on its use.

Another thing that you might want to use ptrepack for is changing the compression filters or compression levels on your existing data for different goals, like checking how this can affect both final size and I/O performance, or getting rid of the optional compressors like LZO or bzip2 in your existing files, in case you want to use them with generic HDF5 tools that do not have support for these filters.

Part II

Complementary modules

CHAPTER

SIX

FILENODE - SIMULATING A FILESYSTEM WITH PYTABLES

6.1 What is filenode?

filenode is a module which enables you to create a PyTables database of nodes which can be used like regular opened files in Python. In other words, you can store a file in a PyTables database, and read and write it as you would do with any other file in Python. Used in conjunction with PyTables hierarchical database organization, you can have your database turned into an open, extensible, efficient, high capacity, portable and metadata-rich filesystem for data exchange with other systems (including backup purposes).

Between the main features of filenode, one can list:

- *Open:* Since it relies on PyTables, which in turn, sits over HDF5 (see *[HDGG1]*), a standard hierarchical data format from NCSA.
- *Extensible:* You can define new types of nodes, and their instances will be safely preserved (as are normal groups, leafs and attributes) by PyTables applications having no knowledge of their types. Moreover, the set of possible attributes for a node is not fixed, so you can define your own node attributes.
- *Efficient:* Thanks to PyTables' proven extreme efficiency on handling huge amounts of data. filenode can make use of PyTables' on-the-fly compression and decompression of data.
- *High capacity:* Since PyTables and HDF5 are designed for massive data storage (they use 64-bit addressing even where the platform does not support it natively).
- *Portable:* Since the HDF5 format has an architecture-neutral design, and the HDF5 libraries and PyTables are known to run under a variety of platforms. Besides that, a PyTables database fits into a single file, which poses no trouble for transportation.
- *Metadata-rich:* Since PyTables can store arbitrary key-value pairs (even Python objects!) for every database node. Metadata may include authorship, keywords, MIME types and encodings, ownership information, access control lists (ACL), decoding functions and anything you can imagine!

6.2 Finding a filenode node

filenode nodes can be recognized because they have a NODE_TYPE system attribute with a 'file' value. It is recommended that you use the `File.get_node_attr()` method of tables.File class to get the NODE_TYPE attribute independently of the nature (group or leaf) of the node, so you do not need to care about.

6.3 filenode - simulating files inside PyTables

The filenode module is part of the nodes sub-package of PyTables. The recommended way to import the module is:

```
>>> from tables.nodes import filenode
```

However, filenode exports very few symbols, so you can import * for interactive usage. In fact, you will most probably only use the NodeType constant and the new_node() and open_node() 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.

new_node() and open_node() 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.open_file('fnode.h5', 'w')
```

6.3.1 Creating a new file node

Creation of a new file node is achieved with the new_node() 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 new_node(); 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.new_node(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).

new_node() 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.

new_node() 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.get_node_attr('/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("This is a test text line.", file=fnode)
>>> print("And this is another one.", file=fnode)
>>> print(file=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 line_separator 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 new_node(), you can open it later by calling open_node(). 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(), open_node() 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.open_node(node, 'a+')
>>> print(repr(fnode.readline()))
'This is a test text line.\\n'
>>> print(fnode.tell())
26
>>> print("This is a new line.", file=fnode)
>>> 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 set_node_attr() method of tables.File to do it directly on the proper node, but filenode offers a much more comfortable way to do it. filenode objects have an attrs property which gives you direct access to their corresponding AttributeSet object.

For instance, let us see how to associate MIME type metadata to our file node:

```
>>> fnode.attrs.content_type = 'text/plain; charset=us-ascii'
```

As simple as A-B-C. You can put nearly anything in an attribute, which opens the way to authorship, keywords, permissions and more. Moreover, there is not a fixed list of attributes. However, you should avoid names in all caps or starting with '_', since PyTables and filenode may use them internally. Some valid examples:

```
>>> fnode.attrs.author = "Ivan Vilata i Balaguer"
>>> fnode.attrs.creation_date = '2004-10-20T13:25:25+0200'
>>> fnode.attrs.keywords_en = ["FileNode", "test", "metadata"]
>>> fnode.attrs.keywords_ca = ["FileNode", "prova", "metadades"]
>>> fnode.attrs.owner = 'ivan'
>>> fnode.attrs.acl = {'ivan': 'rw', '@users': 'r'}
```

You can check that these attributes get stored by running the ptdump command on the host PyTables file.

```
$ ptdump -a fnode.h5:/fnode_test
/fnode_test (EArray(113,)) ''
/fnode_test.attrs (AttributeSet), 14 attributes:
[CLASS := 'EARRAY',
EXTDIM := 0,
FLAVOR := 'numpy',
NODE_TYPE := 'file',
NODE_TYPE_VERSION := 2,
TITLE := '',
VERSION := '1.2',
acl := {'ivan': 'rw', '@users': 'r'},
author := 'Ivan Vilata i Balaguer',
content_type := 'text/plain; charset=us-ascii',
creation_date := '2004-10-20T13:25:25+0200',
keywords_ca := ['FileNode', 'prova', 'metadades'],
keywords_en := ['FileNode', 'test', 'metadata'],
owner := 'ivan']
```

Note that filenode makes no assumptions about the meaning of your metadata, so its handling is entirely left to your needs and imagination.

6.4 Complementary notes

You can use file nodes and PyTables groups to mimic a filesystem with files and directories. Since you can store nearly anything you want as file metadata, this enables you to use a PyTables file as a portable compressed backup, even between radically different platforms. Take this with a grain of salt, since node files are restricted in their naming (only valid Python identifiers are valid); however, remember that you can use node titles and metadata to overcome this limitation. Also, you may need to devise some strategy to represent special files such as devices, sockets and such (not necessarily using filenode).

We are eager to hear your opinion about filenode and its potential uses. Suggestions to improve filenode and create other node types are also welcome. Do not hesitate to contact us!

6.5 Current limitations

filenode is still a young piece of software, so it lacks some functionality. This is a list of known current limitations:

1. Node files can only be opened for read-only or read and append mode. This should be enhanced in the future.
2. Near future?
3. Only binary I/O is supported currently (read/write strings of bytes)
4. There is no universal newline support yet. The only new-line character used at the moment is `\n`. This is likely to be improved in a near future.
5. 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.

See *Filenode Module* for detailed documentation on the filenode interface.

Part III

Appendixes

CHAPTER

SEVEN

SUPPORTED DATA TYPES IN PYTABLES

All PyTables datasets can handle the complete set of data types supported by the NumPy (see *[NUMPY]*) package in Python. The data types for table fields can be set via instances of the Col class and its descendants (see *The Col class and its descendants*), while the data type of array elements can be set through the use of the Atom class and its descendants (see *The Atom class and its descendants*).

PyTables uses ordinary strings to represent its *types*, with most of them matching the names of NumPy scalar types. Usually, a PyTables type consists of two parts: a *kind* and a *precision* in bits. The precision may be omitted in types with just one supported precision (like bool) or with a non-fixed size (like string).

There are eight kinds of types supported by PyTables:

- bool: Boolean (true/false) types. Supported precisions: 8 (default) bits.
- int: Signed integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.
- uint: Unsigned integer types. Supported precisions: 8, 16, 32 (default) and 64 bits.
- float: Floating point types. Supported precisions: 16, 32, 64 (default) bits and extended precision floating point (see *note on floating point types*).
- complex: Complex number types. Supported precisions: 64 (32+32), 128 (64+64, default) bits and extended precision complex (see *note on floating point types*).
- string: Raw string types. Supported precisions: 8-bit positive multiples.
- time: Data/time types. Supported precisions: 32 and 64 (default) bits.
- enum: Enumerated types. Precision depends on base type.

Note: Floating point types.

The half precision floating point data type (float16) and extended precision ones (fload96, float128, complex192, complex256) are only available if numpy supports them on the host platform.

Also, in order to use the half precision floating point type (float16) it is required numpy >= 1.6.0.

The time and enum kinds area little bit special, since they represent HDF5 types which have no direct Python counterpart, though atoms of these kinds have a more-or-less equivalent NumPy data type.

There are two types of time: 4-byte signed integer (time32) and 8-byte double precision floating point (time64). Both of them reflect the number of seconds since the Unix epoch, i.e. Jan 1 00:00:00 UTC 1970. They are stored in memory as NumPy's int32 and float64, respectively, and in the HDF5 file using the H5T_TIME class. Integer times are stored on disk as such, while floating point times are split into two signed integer values representing seconds and microseconds (beware: smaller decimals will be lost!).

PyTables also supports HDF5 H5T_ENUM *enumerations* (restricted sets of unique name and unique value pairs). The NumPy representation of an enumerated value (an Enum, see *The Enum class*) depends on the concrete *base type* used to store the enumeration in the HDF5 file. Currently, only scalar integer values (both signed and unsigned) are supported in enumerations. This restriction may be lifted when HDF5 supports other kinds on enumerated values.

Here you have a quick reference to the complete set of supported data types:

Table 7.1: Data types supported for array elements and tables columns in PyTables.

Type Code	Description	C Type	Size (in bytes)	Python Counterpart
bool	boolean	unsigned char	1	bool
int8	8-bit integer	signed char	1	int
uint8	8-bit unsigned integer	unsigned char	1	int
int16	16-bit integer	short	2	int
uint16	16-bit unsigned integer	unsigned short	2	int
int32	integer	int	4	int
uint32	unsigned integer	unsigned int	4	long
int64	64-bit integer	long long	8	long
uint64	unsigned 64-bit integer	unsigned long long	8	long
float16 [1]	half-precision float	•	2	•
float32	single-precision float	float	4	float
float64	double-precision float	double	8	float
float96 [1] [2]	extended precision float	•	12	•
float128 [1] [2]	extended precision float	•	16	•
complex64	single-precision complex	struct {float r, i;}	8	complex
complex128	double-precision complex	struct {double r, i;}	16	complex
complex192 [1]	extended precision complex	•	24	•
complex256 [1]	extended precision complex	•	32	•
string	arbitrary length string	char[]	•	str
time32	integer time	POSIX's time_t	4	int
time64	floating point time	POSIX's struct timeval	8	float
enum	enumerated value	enum	•	•

CHAPTER

EIGHT

CONDITION SYNTAX

Conditions in PyTables are used in methods related with in-kernel and indexed searches such as `Table.where()` or `Table.read_where()`. They are interpreted using Numexpr, a powerful package for achieving C-speed computation of array operations (see *[NUMEXPR]*).

A condition on a table is just a *string* containing a Python expression involving *at least one column*, and maybe some constants and external variables, all combined with algebraic operators and functions. The result of a valid condition is always a *boolean array* of the same length as the table, where the *i*-th element is true if the value of the expression on the *i*-th row of the table evaluates to true

That is the reason why multidimensional fields in a table are not supported in conditions, since the truth value of each resulting multidimensional boolean value is not obvious. Usually, a method using a condition will only consider the rows where the boolean result is true.

For instance, the condition 'sqrt(x*x + y*y) < 1' applied on a table with x and y columns consisting of floating point numbers results in a boolean array where the *i*-th element is true if (unsurprisingly) the value of the square root of the sum of squares of x and y is less than 1. The sqrt() function works element-wise, the 1 constant is adequately broadcast to an array of ones of the length of the table for evaluation, and the *less than* operator makes the result a valid boolean array. A condition like 'mycolumn' alone will not usually be valid, unless mycolumn is itself a column of scalar, boolean values.

In the previous conditions, mycolumn, x and y are examples of *variables* which are associated with columns. Methods supporting conditions do usually provide their own ways of binding variable names to columns and other values. You can read the documentation of `Table.where()` for more information on that. Also, please note that the names None, True and False, besides the names of functions (see below) *can not be overridden*, but you can always define other new names for the objects you intend to use.

Values in a condition may have the following types:

- 8-bit boolean (bool).
- 32-bit signed integer (int).
- 64-bit signed integer (long).
- 32-bit, single-precision floating point number (float or float32).
- 64-bit, double-precision floating point number (double or float64).
- 2x64-bit, double-precision complex number (complex).
- Raw string of bytes (str).

Nevertheless, if the type passed is not among the above ones, it will be silently upcasted, so you don't need to worry too much about passing supported types, except for the Unsigned 64 bits integer, that cannot be upcasted to any of the supported types.

However, the types in PyTables conditions are somewhat stricter than those of Python. For instance, the *only* valid constants for booleans are True and False, and they are *never* automatically cast to integers. The type strengthening also affects the availability of operators and functions. Beyond that, the usual type inference rules apply.

Conditions support the set of operators listed below:

- Logical operators: &, |, ~.
- Comparison operators: <, <=, ==, !=, >=, >.
- Unary arithmetic operators: -.
- Binary arithmetic operators: +, -, *, /, **, %.

Types do not support all operators. Boolean values only support logical and strict (in)equality comparison operators, while strings only support comparisons, numbers do not work with logical operators, and complex comparisons can only check for strict (in)equality. Unsupported operations (including invalid castings) raise NotImplementedError exceptions.

You may have noticed the special meaning of the usually bitwise operators &, | and ~. Because of the way Python handles the short-circuiting of logical operators and the truth values of their operands, conditions must use the bitwise operator equivalents instead. This is not difficult to remember, but you must be careful because bitwise operators have a *higher precedence* than logical operators. For instance, 'a and b == c' (*a is true AND b is equal to c*) is *not* equivalent to 'a & b == c' (*a AND b is equal to c)*. The safest way to avoid confusions is to *use parentheses* around logical operators, like this: 'a & (b == c)'. Another effect of short-circuiting is that expressions like '0 < x < 1' will *not* work as expected; you should use '(0 < x) & (x < 1)'.

All of this may be solved if Python supported overloadable boolean operators (see PEP 335) or some kind of non-shortcircuiting boolean operators (like C's &&, || and !).

You can also use the following functions in conditions:

- where(bool, number1, number2): number - number1 if the bool condition is true, number2 otherwise.
- {sin,cos,tan}(float|complex): float|complex - trigonometric sine, cosine or tangent.
- {arcsin,arccos,arctan}(float|complex): float|complex - trigonometric inverse sine, cosine or tangent.
- arctan2(float1, float2): float - trigonometric inverse tangent of float1/float2.
- {sinh,cosh,tanh}(float|complex): float|complex - hyperbolic sine, cosine or tangent.
- {arcsinh,arccosh,arctanh}(float|complex): float|complex - hyperbolic inverse sine, cosine or tangent.
- {log,log10,log1p}(float|complex): float|complex - natural, base-10 and log(1+x) logarithms.
- {exp,expm1}(float|complex): float|complex - exponential and exponential minus one.
- sqrt(float|complex): float|complex - square root.
- {abs}(float|complex): float|complex - absolute value.
- {real,imag}(complex): float - real or imaginary part of complex.
- complex(float, float): complex - complex from real and imaginary parts.

CHAPTER

NINE

PYTABLES PARAMETER FILES

PyTables issues warnings when certain limits are exceeded. Those limits are not intrinsic limitations of the underlying software, but rather are proactive measures to avoid large resource consumptions. The default limits should be enough for most of cases, and users should try to respect them. However, in some situations, it can be convenient to increase (or decrease) these limits.

Also, and in order to get maximum performance, PyTables implements a series of sophisticated features, like I/O buffers or different kind of caches (for nodes, chunks and other internal metadata). These features comes with a default set of parameters that ensures a decent performance in most of situations. But, as there is always a need for every case, it is handy to have the possibility to fine-tune some of these parameters.

Because of these reasons, PyTables implements a couple of ways to change the values of these parameters. All the *tunable* parameters live in the `tables/parameters.py`. The user can choose to change them in the parameter files themselves for a global and persistent change. Moreover, if he wants a finer control, he can pass any of these parameters directly to the `tables.open_file()` function, and the new parameters will only take effect in the corresponding file (the defaults will continue to be in the parameter files).

A description of all of the tunable parameters follows. As the defaults stated here may change from release to release, please check with your actual parameter files so as to know your actual default values.

Warning: Changing the next parameters may have a very bad effect in the resource consumption and performance of your PyTables scripts.
Please be careful when touching these!

9.1 Tunable parameters in parameters.py

9.1.1 Recommended maximum values

`tables.parameters.`**`MAX_COLUMNS`** **= 512**
Maximum number of columns in `tables.Table` objects before a `tables.PerformanceWarning` is issued. This limit is somewhat arbitrary and can be increased.

`tables.parameters.`**`MAX_NODE_ATTRS`** **= 4096**
Maximum allowed number of attributes in a node.

`tables.parameters.`**`MAX_GROUP_WIDTH`** **= 16384**
Maximum allowed number of children hanging from a group.

`tables.parameters.`**`MAX_TREE_DEPTH`** **= 2048**
Maximum depth in object tree allowed.

`tables.parameters.`**`MAX_UNDO_PATH_LENGTH`** **= 10240**
Maximum length of paths allowed in undo/redo operations.

9.1.2 Cache limits

`tables.parameters.`**`CHUNK_CACHE_NELMTS`** **= 521**
Number of elements for HDF5 chunk cache.

`tables.parameters.`**`CHUNK_CACHE_PREEMPT`** **= 0.0**
Chunk preemption policy. This value should be between 0 and 1 inclusive and indicates how much chunks that have been fully read are favored for preemption. A value of zero means fully read chunks are treated no differently than other chunks (the preemption is strictly LRU) while a value of one means fully read chunks are always preempted before other chunks.

`tables.parameters.`**`CHUNK_CACHE_SIZE`** **= 2097152**
Size (in bytes) for HDF5 chunk cache.

`tables.parameters.`**`COND_CACHE_SLOTS`** **= 128**
Maximum number of conditions for table queries to be kept in memory.

`tables.parameters.`**`METADATA_CACHE_SIZE`** **= 1048576**
Size (in bytes) of the HDF5 metadata cache.

`tables.parameters.`**`NODE_CACHE_SLOTS`** **= 64**
Maximum number of nodes to be kept in the metadata cache.

It is the number of nodes to be kept in the metadata cache. Least recently used nodes are unloaded from memory when this number of loaded nodes is reached. To load a node again, simply access it as usual. Nodes referenced by user variables and, in general, all nodes that are still open are registered in the node manager and can be quickly accessed even if they are not in the cache.

Negative value means that all the touched nodes will be kept in an internal dictionary. This is the faster way to load/retrieve nodes. However, and in order to avoid a large memory comsumption, the user will be warned when the number of loaded nodes will reach the `-NODE_CACHE_SLOTS` value.

Finally, a value of zero means that any cache mechanism is disabled.

9.1.3 Parameters for the different internal caches

`tables.parameters.`**`BOUNDS_MAX_SIZE`** **= 1048576**
The maximum size for bounds values cached during index lookups.

`tables.parameters.`**`BOUNDS_MAX_SLOTS`** **= 4096**
The maximum number of slots for the BOUNDS cache.

`tables.parameters.`**`ITERSEQ_MAX_ELEMENTS`** **= 1024**
The maximum number of iterator elements cached in data lookups.

`tables.parameters.`**`ITERSEQ_MAX_SIZE`** **= 1048576**
The maximum space that will take ITERSEQ cache (in bytes).

`tables.parameters.`**`ITERSEQ_MAX_SLOTS`** **= 128**
The maximum number of slots in ITERSEQ cache.

`tables.parameters.`**`LIMBOUNDS_MAX_SIZE`** **= 262144**
The maximum size for the query limits (for example, `(lim1, lim2)` in conditions like `lim1 <= col < lim2`) cached during index lookups (in bytes).

`tables.parameters.`**`LIMBOUNDS_MAX_SLOTS`** **= 128**
The maximum number of slots for LIMBOUNDS cache.

`tables.parameters.`**`TABLE_MAX_SIZE`** **= 1048576**
The maximum size for table chunks cached during index queries.

`tables.parameters.`**`SORTED_MAX_SIZE`** **= 1048576**
The maximum size for sorted values cached during index lookups.

`tables.parameters.`**`SORTEDLR_MAX_SIZE`** **= 8388608**
The maximum size for chunks in last row cached in index lookups (in bytes).

`tables.parameters.`**`SORTEDLR_MAX_SLOTS`** **= 1024**
The maximum number of chunks for SORTEDLR cache.

9.1.4 Parameters for general cache behaviour

> **Warning:** The next parameters will not take any effect if passed to the open_file() function, so they can only be changed in a *global* way. You can change them in the file, but this is strongly discouraged unless you know well what you are doing.

`tables.parameters.`**`DISABLE_EVERY_CYCLES`** **= 10**
The number of cycles in which a cache will be forced to be disabled if the hit ratio is lower than the LOWEST_HIT_RATIO (see below). This value should provide time enough to check whether the cache is being efficient or not.

`tables.parameters.`**`ENABLE_EVERY_CYCLES`** **= 50**
The number of cycles in which a cache will be forced to be (re-)enabled, irregardingly of the hit ratio. This will provide a chance for checking if we are in a better scenario for doing caching again.

`tables.parameters.`**`LOWEST_HIT_RATIO`** **= 0.6**
The minimum acceptable hit ratio for a cache to avoid disabling (and freeing) it.

9.1.5 Parameters for the I/O buffer in Leaf objects

`tables.parameters.`**`IO_BUFFER_SIZE`** **= 1048576**
The PyTables internal buffer size for I/O purposes. Should not exceed the amount of highest level cache size in your CPU.

`tables.parameters.`**`BUFFER_TIMES`** **= 100**
The maximum buffersize/rowsize ratio before issuing a *`tables.PerformanceWarning`*.

9.1.6 Miscellaneous

`tables.parameters.`**`EXPECTED_ROWS_EARRAY`** **= 1000**
Default expected number of rows for `EArray` objects.

`tables.parameters.`**`EXPECTED_ROWS_TABLE`** **= 10000**
Default expected number of rows for `Table` objects.

`tables.parameters.`**`PYTABLES_SYS_ATTRS`** **= True**
Set this to `False` if you don’t want to create PyTables system attributes in datasets. Also, if set to `False` the possible existing system attributes are not considered for guessing the class of the node during its loading from disk (this work is delegated to the PyTables’ class discoverer function for general HDF5 files).

`tables.parameters.`**`MAX_NUMEXPR_THREADS`** **= 2**
The maximum number of threads that PyTables should use internally in Numexpr. If *None*, it is automatically set to the number of cores in your machine. In general, it is a good idea to set this to the number of cores in your machine or, when your machine has many of them (e.g. > 8), perhaps stay at 8 at maximum. In general, 2 threads is a good tradeoff.

`tables.parameters.`**`MAX_BLOSC_THREADS`** **= 2**

The maximum number of threads that PyTables should use internally in Blosc. If *None*, it is automatically set to the number of cores in your machine. In general, it is a good idea to set this to the number of cores in your machine or, when your machine has many of them (e.g. > 8), perhaps stay at 8 at maximum. In general, 2 threads is a good tradeoff.

9.1.7 HDF5 driver management

`tables.parameters.`**`DRIVER`** **= None**

The HDF5 driver that should be used for reading/writing to the file.

Following drivers are supported:

- H5FD_SEC2: this driver uses POSIX file-system functions like read and write to perform I/O to a single, permanent file on local disk with no system buffering. This driver is POSIX-compliant and is the default file driver for all systems.
- H5FD_DIRECT: this is the H5FD_SEC2 driver except data is written to or read from the file synchronously without being cached by the system.
- H5FD_WINDOWS: this driver was modified in HDF5-1.8.8 to be a wrapper of the POSIX driver, H5FD_SEC2. This change should not affect user applications.
- H5FD_STDIO: this driver uses functions from the standard C stdio.h to perform I/O to a single, permanent file on local disk with additional system buffering.
- H5FD_CORE: with this driver, an application can work with a file in memory for faster reads and writes. File contents are kept in memory until the file is closed. At closing, the memory version of the file can be written back to disk or abandoned.
- H5FD_SPLIT: this file driver splits a file into two parts. One part stores metadata, and the other part stores raw data. This splitting a file into two parts is a limited case of the Multi driver.

The following drivers are not currently supported:

- H5FD_LOG: this is the H5FD_SEC2 driver with logging capabilities.
- H5FD_FAMILY: with this driver, the HDF5 file's address space is partitioned into pieces and sent to separate storage files using an underlying driver of the user's choice. This driver is for systems that do not support files larger than 2 gigabytes.
- H5FD_MULTI: with this driver, data can be stored in multiple files according to the type of the data. I/O might work better if data is stored in separate files based on the type of data. The Split driver is a special case of this driver.
- H5FD_MPIO: this is the standard HDF5 file driver for parallel file systems. This driver uses the MPI standard for both communication and file I/O.
- H5FD_MPIPOSIX: this parallel file system driver uses MPI for communication and POSIX file-system calls for file I/O.
- H5FD_STREAM: this driver is no longer available.

See also:

the Drivers section of the HDF5 User's Guide for more information.

Note: not all supported drivers are always available. For example the H5FD_WINDOWS driver is not available on non Windows platforms.

If the user try to use a driver that is not available on the target platform a `RuntimeError` is raised.

New in version 3.0.

`tables.parameters.`**`DRIVER_DIRECT_ALIGNMENT`** **= 0**

Specifies the required alignment boundary in memory.

A value of 0 (zero) means to use HDF5 Library's default value.

New in version 3.0.

`tables.parameters.`**`DRIVER_DIRECT_BLOCK_SIZE`** **= 0**

Specifies the file system block size.

A value of 0 (zero) means to use HDF5 Library's default value of 4KB.

New in version 3.0.

`tables.parameters.`**`DRIVER_DIRECT_CBUF_SIZE`** **= 0**

Specifies the copy buffer size.

A value of 0 (zero) means to use HDF5 Library's default value.

New in version 3.0.

`tables.parameters.`**`DRIVER_CORE_INCREMENT`** **= 65536**

Core driver memory increment.

Specifies the increment by which allocated memory is to be increased each time more memory is required.

New in version 3.0.

`tables.parameters.`**`DRIVER_CORE_BACKING_STORE`** **= 1**

Enables backing store for the core driver.

With the H5FD_CORE driver, if the DRIVER_CORE_BACKING_STORE is set to 1 (True), the file contents are flushed to a file with the same name as this core file when the file is closed or access to the file is terminated in memory.

The application is allowed to open an existing file with H5FD_CORE driver. In that case, if the DRIVER_CORE_BACKING_STORE is set to 1 and the flags for `tables.open_file()` is set to H5F_ACC_RDWR, any change to the file contents are saved to the file when the file is closed. If backing_store is set to 0 and the flags for `tables.open_file()` is set to H5F_ACC_RDWR, any change to the file contents will be lost when the file is closed. If the flags for `tables.open_file()` is set to H5F_ACC_RDONLY, no change to the file is allowed either in memory or on file.

New in version 3.0.

`tables.parameters.`**`DRIVER_CORE_IMAGE`** **= None**

String containing an HDF5 file image.

If this oprion is passed to the `tables.open_file()` function then the returned file object is set up using the specified image.

A file image can be retrieved from an existing (and opened) file object using the `tables.File.get_file_image()` method.

Note: requires HDF5 >= 1.8.9.

New in version 3.0.

`tables.parameters.`**`DRIVER_SPLIT_META_EXT`** **= '-m.h5'**

The extension for the metadata file used by the H5FD_SPLIT driver.

If this option is passed to the `tables.openFile()` function along with driver='H5FD_SPLIT', the extension is appended to the name passed as the first parameter to form the name of the metadata file. If the string

'%s' is used in the extension, the metadata file name is formed by replacing '%s' with the name passed as the first parameter instead.

New in version 3.1.

`tables.parameters.`**`DRIVER_SPLIT_RAW_EXT`** **= '-r.h5'**

The extension for the raw data file used by the H5FD_SPLIT driver.

If this option is passed to the `tables.openFile()` function along with driver='H5FD_SPLIT', the extension is appended to the name passed as the first parameter to form the name of the raw data file. If the string '%s' is used in the extension, the raw data file name is formed by replacing '%s' with the name passed as the first parameter instead.

New in version 3.1.

CHAPTER

TEN

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.

10.1 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*).

10.1.1 Usage

For instructions on how to use it, just pass the -h flag to the command:

```
$ ptdump -h
```

to see the message usage:

```
usage: ptdump [-d] [-v] [-a] [-c] [-i] [-R start,stop,step] [-h] file[:nodepath]
    -d -- Dump data information on leaves
    -v -- Dump more metainformation on nodes
    -a -- Show attributes in nodes (only useful when -v or -d are active)
    -c -- Show info of columns in tables (only useful when -v or -d are active)
    -i -- Show info of indexed columns (only useful when -v or -d are active)
    -R RANGE -- Select a RANGE of rows in the form "start,stop,step"
    -h -- Print help on usage
```

Read on for a brief introduction to this utility.

10.1.2 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
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
```

we can see that the file contains just a leaf object called vlarray1, that is an instance of VLArray, has 4 rows, and two filters has been used in order to create it: shuffle and zlib (with a compression level of 1).

Let's say we want more meta-information. Just add the -v (verbose) flag:

```
$ ptdump -v vlarray1.h5
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numpy'
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  atom = StringAtom(itemsize=2, shape=(), dflt='')
  byteorder = 'irrelevant'
  nrows = 3
  flavor = 'python'
```

so we can see more info about the atoms that are the components of the vlarray1 dataset, i.e. they are scalars of type Int32 and with NumPy *flavor*.

If we want information about the attributes on the nodes, we must add the -a flag:

```
$ ptdump -va vlarray1.h5
/ (RootGroup) ''
  /._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'GROUP',
    PYTABLES_FORMAT_VERSION := '2.0',
    TITLE := '',
    VERSION := '1.0']
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numpy'
  /vlarray1._v_attrs (AttributeSet), 3 attributes:
   [CLASS := 'VLARRAY',
    TITLE := 'ragged array of ints',
    VERSION := '1.3']
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  atom = StringAtom(itemsize=2, shape=(), dflt='')
  byteorder = 'irrelevant'
  nrows = 3
  flavor = 'python'
  /vlarray2._v_attrs (AttributeSet), 4 attributes:
   [CLASS := 'VLARRAY',
    FLAVOR := 'python',
    TITLE := 'ragged array of strings',
    VERSION := '1.3']
```

Let's have a look at the real data:

```
$ ptdump -d vlarray1.h5
/ (RootGroup) ''
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[0] [5 6]
[1] [5 6 7]
```

```
[2] [5 6 9 8]
/vlarray2 (VLArray(3,), shuffle, zlib(1)) 'ragged array of strings'
  Data dump:
[0] ['5', '66']
[1] ['5', '6', '77']
[2] ['5', '6', '9', '88']
```

We see here a data dump of the 4 rows in vlarray1 object, in the form of a list. Because the object is a VLA, we see a different number of integers on each row.

Say that we are interested only on a specific *row range* of the /vlarray1 object:

```
ptdump -R2,3 -d vlarray1.h5:/vlarray1
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  Data dump:
[2] [5 6 9 8]
```

Here, we have specified the range of rows between 2 and 4 (the upper limit excluded, as usual in Python). See how we have selected only the /vlarray1 object for doing the dump (vlarray1.h5:/vlarray1).

Finally, you can mix several information at once:

```
$ ptdump -R2,3 -vad vlarray1.h5:/vlarray1
/vlarray1 (VLArray(3,), shuffle, zlib(1)) 'ragged array of ints'
  atom = Int32Atom(shape=(), dflt=0)
  byteorder = 'little'
  nrows = 3
  flavor = 'numpy'
  /vlarray1._v_attrs (AttributeSet), 3 attributes:
   [CLASS := 'VLARRAY',
    TITLE := 'ragged array of ints',
    VERSION := '1.3']
  Data dump:
[2] [5 6 9 8]
```

10.2 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.

10.2.1 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-c
    -h -- Print usage message.
    -v -- Show more information.
    -o -- Overwrite destination file.
```

```
  -R RANGE -- Select a RANGE of rows (in the form "start,stop,step")
      during the copy of \*all* the leaves.  Default values are
      "None,None,1", which means a copy of all the rows.
  --non-recursive -- Do not do a recursive copy. Default is to do it.
  --dest-title=title -- Title for the new file (if not specified,
      the source is copied).
  --dont-create-sysattrs -- Do not create sys attrs (default is to do it).
  --dont-copy-userattrs -- Do not copy the user attrs (default is to do it).
  --overwrite-nodes -- Overwrite destination nodes if they exist. Default is
      to not overwrite them.
  --complevel=(0-9) -- Set a compression level (0 for no compression, which
      is the default).
  --complib=lib -- Set the compression library to be used during the copy.
      lib can be set to "zlib", "lzo", "bzip2" or "blosc".
      Additional compressors for Blosc like "blosc:blosclz",
      "blosc:lz4", "blosc:lz4hc", "blosc:snappy" and
      "blosc:zlib", are supported too.  Defaults to "zlib".
  --shuffle=(0|1) -- Activate or not the shuffling filter (default is active
      if complevel>0).
  --fletcher32=(0|1) -- Whether to activate or not the fletcher32 filter
      (not active by default).
  --keep-source-filters -- Use the original filters in source files. The
      default is not doing that if any of --complevel, --complib, --shuffle
      or --fletcher32 option is specified.
  --chunkshape=("keep"\|"auto"\|int|tuple) -- Set a chunkshape.  A value
      of "auto" computes a sensible value for the chunkshape of the
      leaves copied.  The default is to "keep" the original value.
  --upgrade-flavors -- When repacking PyTables 1.x files, the flavor of
      leaves will be unset. With this, such a leaves will be serialized
      as objects with the internal flavor ('numpy' for 2.x series).
  --dont-regenerate-old-indexes -- Disable regenerating old indexes. The
      default is to regenerate old indexes as they are found.
  --sortby=column -- Do a table copy sorted by the index in "column".
      For reversing the order, use a negative value in the "step" part of
      "RANGE" (see "-R" flag).  Only applies to table objects.
  --checkCSI -- Force the check for a CSI index for the --sortby column.
  --propindexes -- Propagate the indexes existing in original tables.  The
      default is to not propagate them.  Only applies to table objects.
```

Read on for a brief introduction to this utility.

10.2.2 A small tutorial on ptrepack

Imagine that we have ended the tutorial 1 (see the output of examples/tutorial1-1.py), and we want to copy our reduced data (i.e. those datasets that hangs from the /column group) to another file. First, let's remember the content of the examples/tutorial1.h5:

```
$ ptdump tutorial1.h5
/ (RootGroup) 'Test file'
/columns (Group) 'Pressure and Name'
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/detector (Group) 'Detector information'
/detector/readout (Table(10,)) 'Readout example'
```

Now, copy the /columns to other non-existing file. That's easy:

```
$ ptrepack tutorial1.h5:/columns reduced.h5
```

That's all. Let's see the contents of the newly created reduced.h5 file:

```
$ ptdump reduced.h5
/ (RootGroup) ''
/name (Array(3,)) 'Name column selection'
/pressure (Array(3,)) 'Pressure column selection'
```

so, you have copied the children of /columns group into the *root* of the reduced.h5 file.

Now, you suddenly realized that what you intended to do was to copy all the hierarchy, the group /columns itself included. You can do that by just specifying the destination group:

```
$ ptrepack tutorial1.h5:/columns reduced.h5:/columns
$ ptdump reduced.h5
/ (RootGroup) ''
/name (Array(3,)) 'Name column selection'
/pressure (Array(3,)) 'Pressure column selection'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

OK. Much better. But you want to get rid of the existing nodes on the new file. You can achieve this by adding the -o flag:

```
$ ptrepack -o tutorial1.h5:/columns reduced.h5:/columns
$ ptdump reduced.h5
/ (RootGroup) ''
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

where you can see how the old contents of the reduced.h5 file has been overwritten.

You can copy just one single node in the repacking operation and change its name in destination:

```
$ ptrepack tutorial1.h5:/detector/readout reduced.h5:/rawdata
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

where the /detector/readout has been copied to /rawdata in destination.

We can change the filter properties as well:

```
$ ptrepack --complevel=1 tutorial1.h5:/detector/readout reduced.h5:/rawdata
Problems doing the copy from 'tutorial1.h5:/detector/readout' to 'reduced.h5:/rawdata'
The error was --> tables.exceptions.NodeError: destination group \``/\`` already has a node named \``
The destination file looks like:
/ (RootGroup) ''
/rawdata (Table(10,)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
Traceback (most recent call last):
  File "utils/ptrepack", line 3, in ?
```

```
    main()
  File ".../tables/scripts/ptrepack.py", line 349, in main
    stats = stats, start = start, stop = stop, step = step)
  File ".../tables/scripts/ptrepack.py", line 107, in copy_leaf
    raise RuntimeError, "Please check that the node names are not
    duplicated in destination, and if so, add the --overwrite-nodes flag
    if desired."
RuntimeError: Please check that the node names are not duplicated in
destination, and if so, add the --overwrite-nodes flag if desired.
```

Ooops! We ran into problems: we forgot that the /rawdata pathname already existed in destination file. Let's add the –overwrite-nodes, as the verbose error suggested:

```
$ ptrepack --overwrite-nodes --complevel=1 tutorial1.h5:/detector/readout
reduced.h5:/rawdata
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,), shuffle, zlib(1)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
```

you can check how the filter properties has been changed for the /rawdata table. Check as the other nodes still exists.

Finally, let's copy a *slice* of the readout table in origin to destination, under a new group called /slices and with the name, for example, aslice:

```
$ ptrepack -R1,8,3 tutorial1.h5:/detector/readout reduced.h5:/slices/aslice
$ ptdump reduced.h5
/ (RootGroup) ''
/rawdata (Table(10,), shuffle, zlib(1)) 'Readout example'
/columns (Group) ''
/columns/name (Array(3,)) 'Name column selection'
/columns/pressure (Array(3,)) 'Pressure column selection'
/slices (Group) ''
/slices/aslice (Table(3,)) 'Readout example'
```

note how only 3 rows of the original readout table has been copied to the new aslice destination. Note as well how the previously nonexistent slices group has been created in the same operation.

10.3 pt2to3

The PyTables 3.x series now follows PEP 8 coding standard. This makes using PyTables more idiomatic with surrounding Python code that also adheres to this standard. The primary way that the 2.x series was *not* PEP 8 compliant was with respect to variable naming conventions. Approximately 450 API variables were identified and updated for PyTables 3.x.

To ease migration, PyTables ships with a new `pt2to3` command line tool. This tool will run over a file and replace any instances of the old variable names with the 3.x version of the name. This tool covers the overwhelming majority of cases was used to transition the PyTables code base itself! However, it may also accidentally also pick up variable names in 3rd party codes that have *exactly* the same name as a PyTables' variable. This is because `pt2to3` was implemented using regular expressions rather than a fancier AST-based method. By using regexes, `pt2to3` works on Python and Cython code.

`pt2to3` **help:**

```
usage: pt2to3 [-h] [-r] [-p] [-o OUTPUT] [-i] filename

PyTables 2.x -> 3.x API transition tool This tool displays to standard out, so
it is common to pipe this to another file: $ pt2to3 oldfile.py > newfile.py

positional arguments:
  filename              path to input file.

optional arguments:
  -h, --help            show this help message and exit
  -r, --reverse         reverts changes, going from 3.x -> 2.x.
  -p, --no-ignore-previous
                        ignores previous_api() calls.
  -o OUTPUT             output file to write to.
  -i, --inplace         overwrites the file in-place.
```

Note that `pt2to3` only works on a single file, not a a directory. However, a simple BASH script may be written to run `pt2to3` over an entire directory and all sub-directories:

```
#!/bin/bash
for f in $(find .)
do
    echo $f
    pt2to3 $f > temp.txt
    mv temp.txt $f
done
```

Note: **`pt2to3`** uses the `argparse` module that is part of the Python standard library since Python 2.7. Users of Python 2.6 should install `argparse` separately (e.g. via **`pip`**).

The old APIs and variable names will continue to be supported for the short term, where possible. (The major backwards incompatible changes come from the renaming of some function and method arguments and keyword arguments.) Using the 2.x APIs in the 3.x series, however, will issue warnings. The following is the release plan for the warning types:

- 3.0 - PendingDeprecationWarning
- 3.1 - DeprecationWarning
- >=3.2 - Remove warnings, previous_api(), and _past.py; keep pt2to3,

The current plan is to maintain the old APIs for at least 2 years, though this is subject to change.

CHAPTER

ELEVEN

PYTABLES FILE FORMAT

PyTables has a powerful capability to deal with native HDF5 files created with another tools. However, there are situations were you may want to create truly native PyTables files with those tools while retaining fully compatibility with PyTables format. That is perfectly possible, and in this appendix is presented the format that you should endow to your own-generated files in order to get a fully PyTables compatible file.

We are going to describe the *2.0 version of PyTables file format* (introduced in PyTables version 2.0). As time goes by, some changes might be introduced (and documented here) in order to cope with new necessities. However, the changes will be carefully pondered so as to ensure backward compatibility whenever is possible.

A PyTables file is composed with arbitrarily large amounts of HDF5 groups (Groups in PyTables naming scheme) and datasets (Leaves in PyTables naming scheme). For groups, the only requirements are that they must have some *system attributes* available. By convention, system attributes in PyTables are written in upper case, and user attributes in lower case but this is not enforced by the software. In the case of datasets, besides the mandatory system attributes, some conditions are further needed in their storage layout, as well as in the datatypes used in there, as we will see shortly.

As a final remark, you can use any filter as you want to create a PyTables file, provided that the filter is a standard one in HDF5, like *zlib*, *shuffle* or *szip* (although the last one can not be used from within PyTables to create a new file, datasets compressed with szip can be read, because it is the HDF5 library which do the decompression transparently).

11.1 Mandatory attributes for a File

The File object is, in fact, an special HDF5 *group* structure that is *root* for the rest of the objects on the object tree. The next attributes are mandatory for the HDF5 *root group* structure in PyTables files:

- *CLASS*: This attribute should always be set to ‘GROUP’ for group structures.
- *PYTABLES_FORMAT_VERSION*: It represents the internal format version, and currently should be set to the ‘2.0’ string.
- *TITLE*: A string where the user can put some description on what is this group used for.
- *VERSION*: Should contains the string ‘1.0’.

11.2 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’.

11.3 Optional attributes for a Group

The next attributes are optional for *group* structures:

- *FILTERS*: When present, this attribute contains the filter properties (a Filters instance, see section *The Filters class*) that may be inherited by leaves or groups created immediately under this group. This is a packed 64-bit integer structure, where
  - *byte 0* (the least-significant byte) is the compression level (complevel).
  - *byte 1* is the compression library used (complib): 0 when irrelevant, 1 for Zlib, 2 for LZO and 3 for Bzip2.
  - *byte 2* indicates which parameterless filters are enabled (shuffle and fletcher32): bit 0 is for *Shuffle* while bit 1 is for*Fletcher32*.
  - other bytes are reserved for future use.

11.4 Mandatory attributes, storage layout and supported data types for Leaves

This depends on the kind of Leaf. The format for each type follows.

11.4.1 Table format

Mandatory attributes

The next attributes are mandatory for *table* structures:

- *CLASS*: Must be set to 'TABLE'.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '2.6'.
- *FIELD_X_NAME*: It contains the names of the different fields. The X means the number of the field, zero-based (beware, order do matter). You should add as many attributes of this kind as fields you have in your records.
- *FIELD_X_FILL*: It contains the default values of the different fields. All the datatypes are supported natively, except for complex types that are currently serialized using Pickle. The X means the number of the field, zero-based (beware, order do matter). You should add as many attributes of this kind as fields you have in your records. These fields are meant for saving the default values persistently and their existence is optional.
- *NROWS*: This should contain the number of *compound* data type entries in the dataset. It must be an *int* data type.

Storage Layout

A Table has a *dataspace* with a *1-dimensional chunked* layout.

Datatypes supported

The datatype of the elements (rows) of Table must be the H5T_COMPOUND *compound* data type, and each of these compound components must be built with only the next HDF5 data types *classes*:

- *H5T_BITFIELD*: This class is used to represent the Bool type. Such a type must be build using a H5T_NATIVE_B8 datatype, followed by a HDF5 H5Tset_precision call to set its precision to be just 1 bit.
- ***H5T_INTEGER*: This includes the next data types:**
  - *H5T_NATIVE_SCHAR*: This represents a *signed char* C type, but it is effectively used to represent an Int8 type.
  - *H5T_NATIVE_UCHAR*: This represents an *unsigned char* C type, but it is effectively used to represent an UInt8 type.
  - *H5T_NATIVE_SHORT*: This represents a *short* C type, and it is effectively used to represent an Int16 type.
  - *H5T_NATIVE_USHORT*: This represents an *unsigned short* C type, and it is effectively used to represent an UInt16 type.
  - *H5T_NATIVE_INT*: This represents an *int* C type, and it is effectively used to represent an Int32 type.
  - *H5T_NATIVE_UINT*: This represents an *unsigned int* C type, and it is effectively used to represent an UInt32 type.
  - *H5T_NATIVE_LONG*: This represents a *long* C type, and it is effectively used to represent an Int32 or an Int64, depending on whether you are running a 32-bit or 64-bit architecture.
  - *H5T_NATIVE_ULONG*: This represents an *unsigned long* C type, and it is effectively used to represent an UInt32 or an UInt64, depending on whether you are running a 32-bit or 64-bit architecture.
  - *H5T_NATIVE_LLONG*: This represents a *long long* C type (__int64, if you are using a Windows system) and it is effectively used to represent an Int64 type.
  - *H5T_NATIVE_ULLONG*: This represents an *unsigned long long* C type (beware: this type does not have a correspondence on Windows systems) and it is effectively used to represent an UInt64 type.
- ***H5T_FLOAT*: This includes the next datatypes:**
  - *H5T_NATIVE_FLOAT*: This represents a *float* C type and it is effectively used to represent an Float32 type.
  - *H5T_NATIVE_DOUBLE*: This represents a *double* C type and it is effectively used to represent an Float64 type.
- ***H5T_TIME*: This includes the next datatypes:**
  - *H5T_UNIX_D32*: This represents a POSIX *time_t* C type and it is effectively used to represent a ‘Time32’ aliasing type, which corresponds to an Int32 type.
  - *H5T_UNIX_D64*: This represents a POSIX *struct timeval* C type and it is effectively used to represent a ‘Time64’ aliasing type, which corresponds to a Float64 type.
- *H5T_STRING*: The datatype used to describe strings in PyTables is H5T_C_S1 (i.e. a *string* C type) followed with a call to the HDF5 H5Tset_size() function to set their length.
- *H5T_ARRAY*: This allows the construction of homogeneous, multidimensional arrays, so that you can include such objects in compound records. The types supported as elements of H5T_ARRAY data types are the ones described above. Currently, PyTables does not support nested H5T_ARRAY types.
- *H5T_COMPOUND*: This allows the support for datatypes that are compounds of compounds (this is also known as *nested types* along this manual).

  This support can also be used for defining complex numbers. Its format is described below:

  The H5T_COMPOUND type class contains two members. Both members must have the H5T_FLOAT atomic datatype class. The name of the first member should be “r” and represents the real part. The name of the second member should be “i” and represents the imaginary part. The *precision* property of both of the

H5T_FLOAT members must be either 32 significant bits (e.g. H5T_NATIVE_FLOAT) or 64 significant bits (e.g. H5T_NATIVE_DOUBLE). They represent Complex32 and Complex64 types respectively.

11.4.2 Array format

Mandatory attributes

The next attributes are mandatory for *array* structures:

- *CLASS*: Must be set to 'ARRAY'.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '2.3'.

Storage Layout

An Array has a *dataspace* with a *N-dimensional contiguous* layout (if you prefer a *chunked* layout see EArray below).

Datatypes supported

The elements of Array must have either HDF5 *atomic* data types or a *compound* data type representing a complex number. The atomic data types can currently be one of the next HDF5 data type *classes*: H5T_BITFIELD, H5T_INTEGER, H5T_FLOAT and H5T_STRING. The H5T_TIME class is also supported for reading existing Array objects, but not for creating them. See the Table format description in *Table format* for more info about these types.

In addition to the HDF5 atomic data types, the Array format supports complex numbers with the H5T_COMPOUND data type class. See the Table format description in *Table format* for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed in Array objects.

11.4.3 CArray format

Mandatory attributes

The next attributes are mandatory for *CArray* structures:

- *CLASS*: Must be set to 'CARRAY'.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.0'.

Storage Layout

An CArray has a *dataspace* with a *N-dimensional chunked* layout.

Datatypes supported

The elements of CArray must have either HDF5 *atomic* data types or a *compound* data type representing a complex number. The atomic data types can currently be one of the next HDF5 data type *classes*: H5T_BITFIELD, H5T_INTEGER, H5T_FLOAT and H5T_STRING. The H5T_TIME class is also supported for reading existing CArray objects, but not for creating them. See the Table format description in *Table format* for more info about these types.

In addition to the HDF5 atomic data types, the CArray format supports complex numbers with the H5T_COMPOUND data type class. See the Table format description in *Table format* for more info about this special type.

You should note that H5T_ARRAY class datatypes are not allowed yet in Array objects.

11.4.4 EArray format

Mandatory attributes

The next attributes are mandatory for *earray* structures:

- *CLASS*: Must be set to 'EARRAY'.
- *EXTDIM*: (*Integer*) Must be set to the extendable dimension. Only one extendable dimension is supported right now.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.3'.

Storage Layout

An EArray has a *dataspace* with a *N-dimensional chunked* layout.

Datatypes supported

The elements of EArray are allowed to have the same data types as for the elements in the Array format. They can be one of the HDF5 *atomic* data type *classes*: H5T_BITFIELD, H5T_INTEGER, H5T_FLOAT, H5T_TIME or H5T_STRING, see the Table format description in *Table format* for more info about these types. They can also be a H5T_COMPOUND datatype representing a complex number, see the Table format description in *Table format*.

You should note that H5T_ARRAY class data types are not allowed in EArray objects.

11.4.5 VLArray format

Mandatory attributes

The next attributes are mandatory for *vlarray* structures:

- *CLASS*: Must be set to 'VLARRAY'.
- *PSEUDOATOM*: This is used so as to specify the kind of pseudo-atom (see *VLArray format*) for the VLArray. It can take the values 'vlstring', 'vlunicode' or 'object'. If your atom is not a pseudo-atom then you should not specify it.
- *TITLE*: A string where the user can put some description on what is this dataset used for.
- *VERSION*: Should contain the string '1.3'.

Storage Layout

An 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 *Table format*. That includes the classes:

- H5T_BITFIELD
- H5T_INTEGER
- H5T_FLOAT
- H5T_TIME
- H5T_STRING
- H5T_ARRAY

They can also be a H5T_COMPOUND data type representing a complex number, see the Table format description in *Table format* for a detailed description.

You should note that this does not include another VL datatype, or a compound datatype that does not fit the description of a complex number. Note as well that, for object and vlstring pseudo-atoms, the base for the VL datatype is always a H5T_NATIVE_UCHAR (H5T_NATIVE_UINT for vlunicode). That means that the complete row entry in the dataset has to be used in order to fully serialize the object or the variable length string.

11.5 Optional attributes for Leaves

The next attributes are optional for *leaves*:

- *FLAVOR*: This is meant to provide the information about the kind of object kept in the Leaf, i.e. when the dataset is read, it will be converted to the indicated flavor. It can take one the next string values:
  - *“numpy”*: Read data (structures arrays, arrays, records, scalars) will be returned as NumPy objects.
  - *“python”*: Read data will be returned as Python lists, tuples, or scalars.

CHAPTER TWELVE

BIBLIOGRAPHY

[HDFG1] The HDF Group. What is HDF5?. Concise description about HDF5 capabilities and its differences from earlier versions (HDF4). http://www.hdfgroup.org/HDF5/whatishdf5.html.

[HDFG2] The HDF Group. Introduction to HDF5. Introduction to the HDF5 data model and programming model. http://www.hdfgroup.org/HDF5/doc/H5.intro.html.

[HDFG3] The HDF Group. The HDF5 table programming model. Examples on using HDF5 tables with the C API. http://www.hdfgroup.org/HDF5/Tutor/h5table.html.

[MERTZ] David Mertz. Objectify. On the 'Pythonic' treatment of XML documents as objects(II). Article describing XML Objectify, a Python module that allows working with XML documents as Python objects. Some of the ideas presented here are used in PyTables. http://gnosis.cx/publish/programming/xml_matters_2.html.

[CYTHON] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, and Greg Ewing. Cython. A language that makes writing C extensions for the Python language as easy as Python itself. http://www.cython.org.

[NUMPY] Travis Oliphant and et al. NumPy. Scientific Computing with Numerical Python. The latest and most powerful re-implementation of Numeric to date. It implements all the features that can be found in Numeric and numarray, plus a bunch of new others. In general, it is more efficient as well. http://www.numpy.org.

[NUMEXPR] David Cooke, Francesc Alted, and et al. Numexpr. Fast evaluation of array expressions by using a vector-based virtual machine. It is an enhaced computing kernel that is generally faster (between 1x and 10x, depending on the kind of operations) than NumPy at evaluating complex array expressions. http://code.google.com/p/numexpr.

[ZLIB] JeanLoup Gailly and Mark Adler. zlib. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. A standard library for compression purposes. http://www.gzip.org/zlib/.

[LZO] Markus F Oberhumer. LZO. A data compression library which is suitable for data de-/compression in real-time. It offers pretty fast compression and decompression with reasonable compression ratio. http://www.oberhumer.com/opensource/.

[BZIP2] Julian Seward. bzip2. A high performance lossless compressor. It offers very high compression ratios within reasonable times. http://www.bzip.org/.

[BLOSC] Francesc Alted. Blosc. A blocking, shuffling and loss-less compression library. A compressor designed to transmit data from memory to CPU (and back) faster than a plain memcpy(). http://www.blosc.org/.

[GNUWIN32] Alexis Wilke, Jerry S., Kees Zeelenberg, and Mathias Michaelis. GnuWin32. GNU (and other) tools ported to Win32. GnuWin32 provides native Win32-versions of GNU tools, or tools with a similar open source licence. http://gnuwin32.sourceforge.net/.

[PSYCO] Armin Rigo. Psyco. A Python specializing compiler. Run existing Python software faster, with no change in your source. http://psyco.sourceforge.net.

[SCIPY1] Konrad Hinsen. Scientific Python. Collection of Python modules useful for scientific computing. http://dirac.cnrs-orleans.fr/ScientificPython.

[SCIPY2] Eric Jones, Travis Oliphant, Pearu Peterson, and et al. SciPy. Scientific tools for Python. SciPy supplements the popular Numeric module, gathering a variety of high level science and engineering modules together as a single package. http://www.scipy.org.

[OPTIM] Francesc Alted and Ivan Vilata. Optimization of file openings in PyTables. This document explores the savings of the opening process in terms of both CPU time and memory, due to the adoption of a LRU cache for the nodes in the object tree. http://www.pytables.org/docs/NewObjectTreeCache.pdf.

[OPSI] Francesc Alted and Ivan Vilata. OPSI: The indexing system of PyTables 2 Professional Edition. Exhaustive description and benchmarks about the indexing engine that comes with PyTables Pro. http://www.pytables.org/docs/OPSI-indexes.pdf.

[VITABLES] Vicent Mas. ViTables. A GUI for PyTables/HDF5 files. It is a graphical tool for browsing and editing files in both PyTables and HDF5 formats. http://vitables.org.

[GIT] Git is a free and open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency http://git-scm.com.

[SPHINX] Sphinx is a tool that makes it easy to create intelligent and beautiful documentation, written by Georg Brandl and licensed under the BSD license http://sphinx-doc.org.

Symbols