Optimization of file openings in PyTables

Francesc Altet, Ivan Vilata

(Dated: October, 14th, 2005)

Before release 1.2, PyTables took a considerable time for opening HDF5 files having a large number of datasets. This was due to the fact that PyTables needed to create an object tree, with the meta-information of all the datasets (and groups), completely in-memory. This meant that using PyTables with files with more than a few tens of thousands datasets was unaffordable for interactive processing.

With version 1.2, we introduced a brand-new cache for the object tree, with a **LRU** (Least Recently Used) algorithm for the discarding of old objects. This effectively removes the need to load all the metadata in opening time, while retaining full backward compatibility with past versions of PyTables.

This document explores the savings of the opening process in terms of both CPU time and memory consumptions. Also, we review how a handful of common user-cases (like browsing the entire hierarchy and read metadata) are affected by this new **LRU** cache. Finally, we also have checked how the new metadata cache introduced in HDF5 1.7.x (development version) would affect performance and memory consumption with different versions of PyTables.

Contents

I. EXPERIMENTAL SET-UP

In order to assess the improvement in the opening times, we have created three different files with the same number of nodes (3000 each one) but with different distributions in the hierarchy:

- **LR** (*Low Ratio*): In this file there are just 3 hierarchical levels, with 1 group on each level and with 1000 datasets (homogeneous arrays) on each group. The group/dataset ratio is 0.001, which is quite low and hence his name.

- **MR** (*Medium Ratio*): Here, the file has 3 hierarchical levels but with 10 groups on each level, having 100 datasets (homogeneous arrays) on each group. The group/dataset ratio is 0.01, which we consider as a medium ratio. This can be considered a representative sample of typical HDF5 files.

- **HR** (*High Ratio*): This time, the file has 30 hierarchical levels with 10 groups on each level, having 10 datasets each group. In this case, the group/dataset ratio is roughly 0.1.

All these files have been created with the file *bench/create-large-number-of-objects.py* (it can be found in the PyTables distribution) that can be easily used to create files with an entirely different distribution, if so desired.

We also have written a benchmarking program to check the different versions of PyTables (see later). It can be found in *bench/open_close-bench2.py*, and it has many interesting features, like checking different common user cases (see later), running the tests in different sub-process so as to not pollute the timings and memory counters (which depend on the entire process), and being able to automatically profile the code by just passing a flag to it.

All the benchmarks have been done in a laptop with a Pentium4 @ 2 GHz and with a hard disk @ 4200 RPM. This is a relatively slow machine nowadays, so you can expect that the times in modern systems would be shortened by a factor of 2 or more. However, the memory usage would be similar, except if you are using 64-bit architectures, where it can be significantly higher (up to 2x).

Also, we have used the next software versions:

- GNU/Linux Debian (Sid)

- Python 2.4.1

- numarray 1.3.3

- HDF5 1.6.4 (production version) and 1.7.51 (development version)

- PyTables 1.1, 1.1.1 and 1.2

II. EXPERIMENT GOALS

First of all, our priority is to check whether the new LRU cache implementation for PyTables does accelerate the opening of HDF5 files and how much. To this effect, we have run the benchmarks in three different versions of PyTables, namely:

- *PyTables 1.1*: This is the latest version previous to the starting of the optimization of file opening tasks. This version builds the entire object tree in the *classic* way and it will be our reference to compare the next optimizations that have been done.

- *PyTables 1.1.1*: This is the latest production version at the time as of this writing. On it, we have applied different optimizations to the *classic* method of openings (i.e., the LRU cache is not there yet).

- *PyTables 1.2*: This contains the optimizations included in 1.1.1 plus our LRU cache implementation for the object tree. At this time, PyTables 1.2 is in the last stages of testing and will be publicly released rather soon.

Besides of openings, we also want to track down if the new **LRU** cache would perform well for some common user cases. We have identified and tested the next user cases:

- **open_close**: This only opens the file and closes it immediately after. The code checked out is simple:

```
fileh=tables.openFile(filename)
fileh.close()
```

- **only_open**: This opens the file and gets times and memory usage before closing it. It's a good measure of how much time would wait an user in interactive mode before being able to access the file contents. The code:

```
fileh=tables.openFile(filename)
```

- **full_browse**: This opens the file, walks all the nodes in there without doing anything, and closes the file. Here is the code:

```
fileh=tables.openFile(filename)
for node in fileh:
    pass
fileh.close()
```

- **partial_browse**: Opens the file, walks on just one hierarchical level and closes the file. The code:

```
fileh=tables.openFile(filename)
for node in fileh.root.ngroup0.ngroup1:
    pass
fileh.close()
```

- **full_browse_attrs**: Opens the file, walks all the nodes in the file and reads one attribute on each. Code follows:

```
fileh=tables.openFile(filename)
for node in fileh:
    # Access to an attribute
    klass = node._v_attrs.CLASS
fileh.close()
```

- **partial_browse_attrs**: Opens the file, walks on just one hierarchical level, reads one attribute on each node found and closes the file. The code reads:

```
fileh=tables.openFile(filename)
for node in fileh.root.ngroup0.ngroup1:
    # Access to an attribute
    klass = node._v_attrs.CLASS
fileh.close()
```

- **open_group**: Opens a file, access to a group in the second level directly, reads an attribute of it and closes the file. The code:

```
fileh=tables.openFile(filename)
group = fileh.root.ngroup0.ngroup1
    # Access to an attribute
    klass = group._v_attrs.CLASS
fileh.close()
```

- **open_leaf**: Opens a file, access to a dataset in the third level directly, reads an attribute of it and closes the file. The code follows:

```
fileh=tables.openFile(filename)
leaf = fileh.root.ngroup0.ngroup1.array9
    # Access to an attribute
    klass = leaf._v_attrs.CLASS
fileh.close()
```

III. RESULTS OF THE EXPERIMENTS

In this section we will study, in terms of speed and memory consumption, the improvements due to the optimizations introduced in PyTables 1.1.1 and 1.2. First we will use the production release of HDF5 (1.6.4). Then we will also use the development version of HDF5 (1.7.51 as of this writing), paying special attention to its new cache for metadata, and how it affects the PyTables performance and memory usage.

A. Comparing the optimization of classic method and the new LRU cache (using HDF5 1.6.4)

We will start presenting the timing results for PyTables 1.1, 1.1.1 and 1.2 for LR, MR and HR files (see above). Note that we will use HDF5 1.6.4 for the next benchmarks, while HDF5 1.7.51 test will be introduced later on.

In Figure 1 we can observe how optimizations made in PyTables 1.1.1 have more than doubled the speed of opening files. In fact, it has almost doubled the speed for all the cases, except for *full_browse_attrs*, where the bottleneck is the access to the attributes; in PyTables 1.1, the attribute **CLASS** (that which is read for each node) is already cached in the object tree, so there is little cost in doing that. However, one of the optimizations in 1.1.1 was deferring the creation of the attribute cache until it was really necessary (this is actually called *lazy* cache creation) in order to make the opening faster, so it has to read the attribute and put it in the newly created cache. This is why the *full_browse_attrs* test is only around 20 % faster in 1.1.1 than 1.1.

The most interesting thing, however, is that the new **LRU** cache allows to open files with thousands of nodes in just a fraction of second, just as intended. However, you can see that if you have to browse the complete hierarchy, the times can become even *worse* than version 1.1. This was also somewhat expected, because every node in the file has to be visited, settled into the cache and then removed when the cache eventually gets full (currently only 256 objects are simultaneously in the cache). It is also worth to note that accessing single nodes (groups or leaves) in the middle of the hierarchy with 1.2 also takes a tiny fraction of the time than with 1.1 or 1.1.1, which was another of our goals.

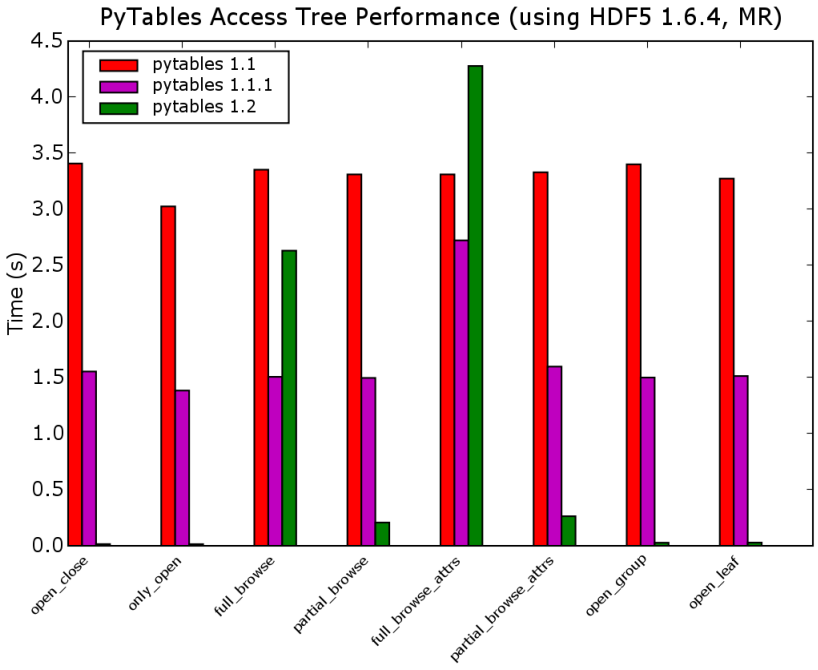

Figure 1: PyTables times for versions 1.1, 1.1.1 and 1.2

Another remarkable fact is that, for accessing the **complete** tree (both in *full_browse* and *full_browse_attrs*), PyTables 1.2 takes more time than accessing this info in the *classical* way (i.e. PyTables 1.1.1); this is particularly evident in the *full_browse_attrs* test where PyTables 1.2 takes 4.28 s while PyTables 1.1.1 takes just 2.72 s. A profiling on this test, shows up what is happening. For example, for PyTables 1.1.1, the profile looks like:

```
225690 function calls in 5.770 CPU seconds

Ordered by:  internal time, call count

ncalls tottime percall cumtime percall filename:lineno(function)

12137 0.790 0.000 0.790 0.000 :0(_g_getSysAttr)

 3000 0.770 0.000 0.770 0.000 :0(_openArray)

12137 0.550 0.000 1.480 0.000 AttributeSet.py:169(__getattr__)

 3034 0.430 0.000 2.430 0.001 AttributeSet.py:120(__init__)

 3034 0.230 0.000 0.230 0.000 :0(_g_listAttr)

21344 0.230 0.000 0.230 0.000 :0(isinstance)

24274 0.190 0.000 0.190 0.000 AttributeSet.py:68(issysattrname)

 3000 0.160 0.000 0.160 0.000 :0(read_g_leaf_attr)
```

While the profile for PyTables 1.2 looks like:

```
1144620 function calls (1141847 primitive calls) in 17.880 CPU seconds

Ordered by:  internal time, call count

ncalls tottime percall cumtime percall filename:lineno(function)

179948 1.890 0.000 2.780 0.000 lrucache.py:106(__cmp__)

 42581 1.860 0.000 8.730 0.000 File.py:889(getNode)

 42547 1.110 0.000 5.900 0.000 File.py:844(_getNode)

179948 0.890 0.000 0.890 0.000 :0(cmp)

 42292 0.830 0.000 1.090 0.000 File.py:228(__getitem__)

100480 0.760 0.000 0.760 0.000 :0(isinstance)

   524 0.710 0.001 2.740 0.005 :0(heapify)

 12137 0.640 0.000 6.080 0.001 AttributeSet.py:205(__getattr__)

 36411 0.560 0.000 4.600 0.000 AttributeSet.py:119(_g_getnode)
```

We can see how time consumption in PyTables 1.2 is dominated by the methods related with accessing the nodes through the new **lrucache** module (`lrucache.py:__cmp__`, `File.py:getNode`, `0:cmp`, `heapify`,...), while dominating times in PyTables 1.1.1 are the reading of attributes and the opening of nodes (just as expected). Although this can be considered as something natural (the existence of a cache in situations where all the data is read just once, always penalize the access times), we feel that this time can be further optimized by converting some of the most consuming methods into Pyrex.

Now, let's have a look at the memory consumption with the same tests discussed above:

In Figure 2, we see that optimizations in 1.1.1 saved around 7 MB of memory compared with 1.1, in part because the object tree that cached less things (mostly the attribute cache). However, when asking for all the attributes, all of them get loaded in-memory in 1.1.1, so using approximately the same memory than 1.1.

Once again, we see that the **LRU** cache in 1.2 really makes a good job when opening a file, as it uses a mere 6 MB than can be attributed to start-up data structures in both PyTables and HDF5. On another hand, when a full browse of the nodes in the hierarchy is done, memory consumption starts to grow noticeably; and still more when attributes are asked for. This noticeable consumption may be due probably to the different caches that are used inside PyTables, HDF5 and also Python interpreter itself. Despite of this, one can see how PyTables 1.2 is still saving almost 10 MB in the worst of the cases (more if HDF5 1.7.51 is used, see later) and 30 MB in the best case, which is quite a few.

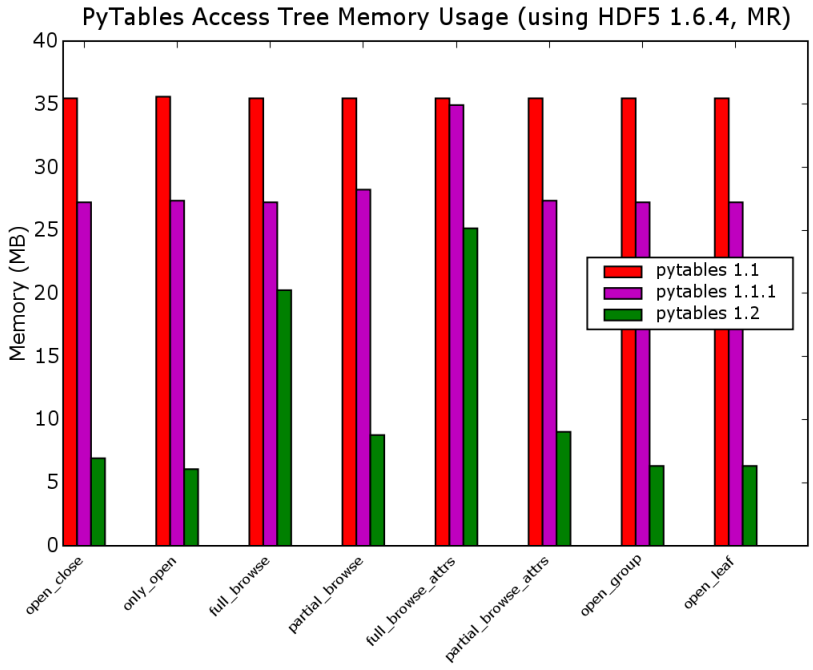

Figure 2: Memory usage for versions 1.1, 1.1.1 and 1.2

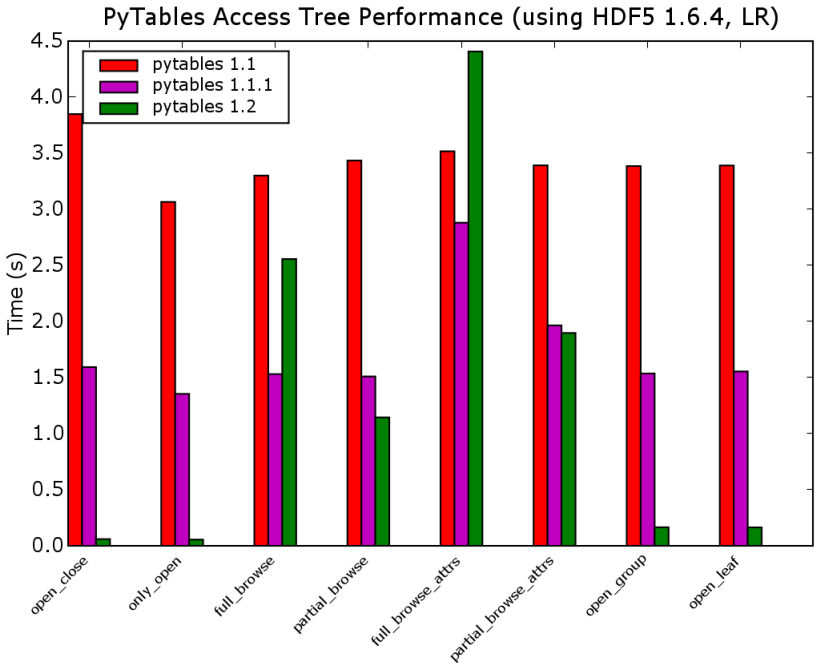

Figure 3: Access times with an LR file

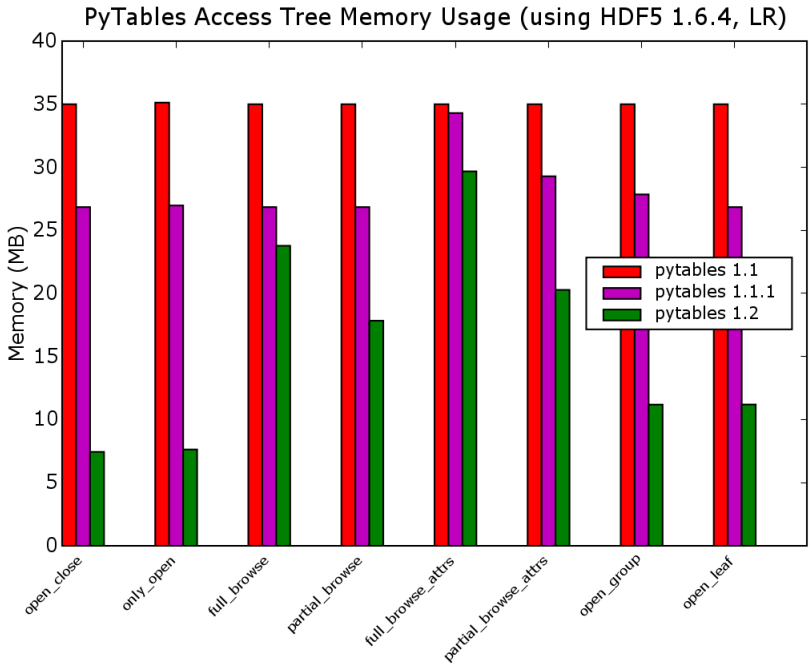

Figure 4: Memory usage with an LR file

Next, we will present graphs for files with a ***L**ow **R**atio* of groups/leaves (figures 3 and 4).

In Figure 3, we can see how for the LR file, PyTables 1.2 takes more time than opening a MR file. More precisely, opening a LR file takes 55 ms while opening a MR file takes 15 ms (10 ms for a HR file), that is, 4x more. However, both times are small enough, and one follows that the introduction of LRU cache enables PyTables the opening of files in less that 0.1 s for the vast majority of files, which should be **more than enough for interactive work**.

We will end this section by showing the Figures when accessing to a file with a ***H**igh **R**atio* of groups/leaves (figures 5 and 6).

In Figure 5, we can see how PyTables 1.2 performs slightly better for opening HR files (10 ms) than opening MR files (15 ms). This is easily explained because the HR file has fewer nodes (20) in the root level than the MR file (110). Again, the most important thing to note is that opening times in PyTables depends very little on how many nodes hangs directly from the root group.

Regarding memory usage, there can be seen a trend showing that PyTables 1.2 need less memory as the *group/leaves* ratio grows (although, again, this does not depends very much on such ratio).

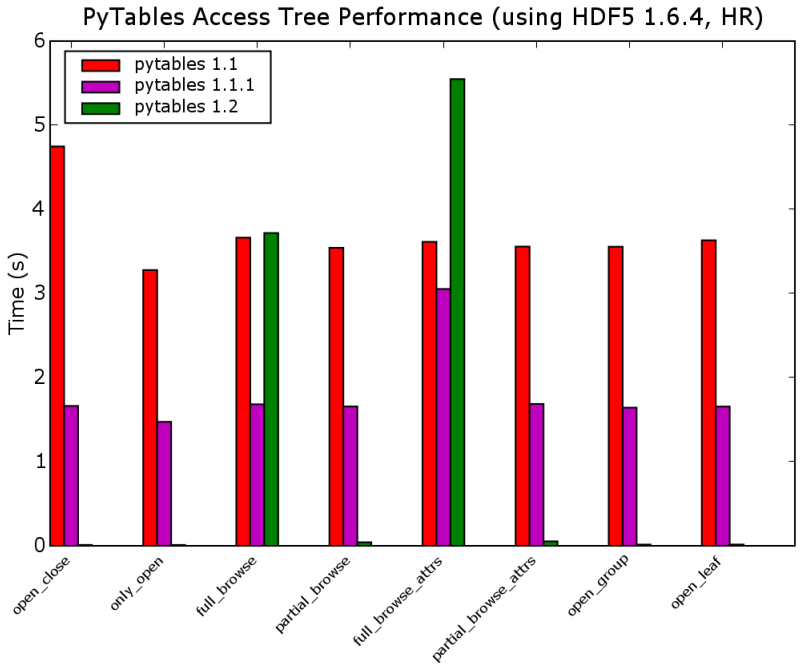

Figure 5: Access times with an HR file

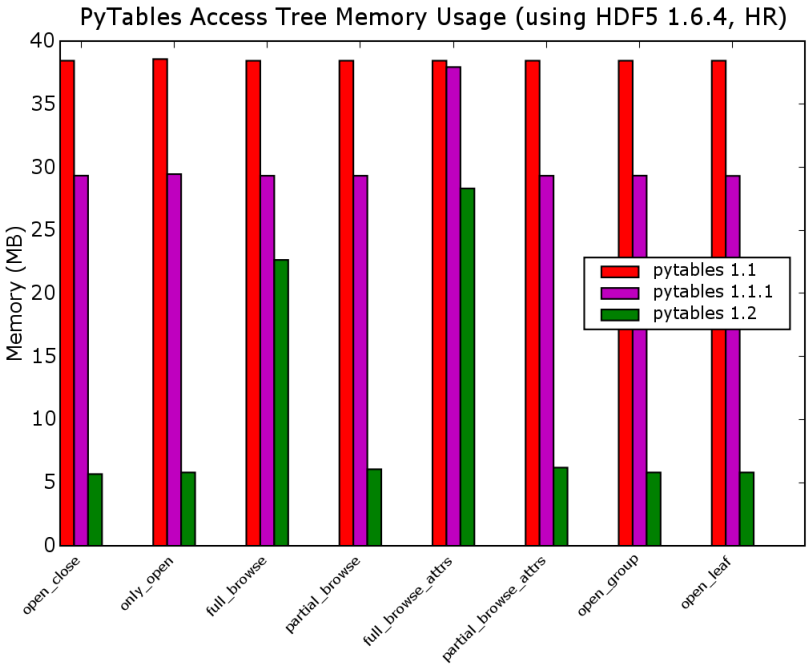

Figure 6: Memory usage with an HR file

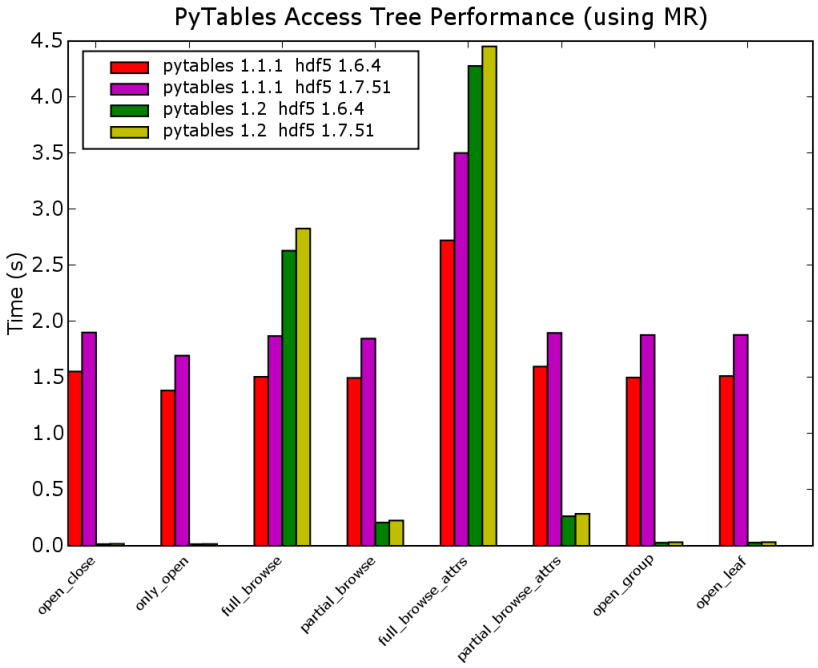

Figure 7: Performance comparison of HDF5 1.6.4 vs 1.7.51

B. Looking at the effects of using HDF5 1.7.51 (development version) instead of 1.6.4.

Forthcoming HDF5 1.8 will wear a new metadata cache that is expected to work much better than the existing one in HDF5 1.6.4. As the cache for metadata is heavily used by some of the benchmarks discussed here, it would be useful to look how it performs in such cases. To this end, we will use the HDF5 1.7.51 version that already has a quite definitive version of the cache that will appear in HDF5 1.8 release.

Besides, this new metadata cache size is configurable by the user, so we have made several runs with different values of the cache size, namely 1 MB (the default), 4 MB and 16 MB.

1. Performance effects

Let's start by looking how performance (Figure 7) is affected by using HDF5 1.7.51 (with default setting for the cache) instead of HDF5 1.6.4.

As we can see in Figure 7, the use of HDF5 1.7.51 seems to affect much more to PyTables 1.1.1 (remember, this is the version with the optimization of the *classic* opening method) than to PyTables 1.2. In particular, the *full_browse* and *full_browse_attrs* are up to a 30% slower (when using HDF5 1.7.51) in the case of PyTables 1.1.1, and just up to a 5% slower for PyTables 1.2. As HDF5 1.7.51 is the precursor of

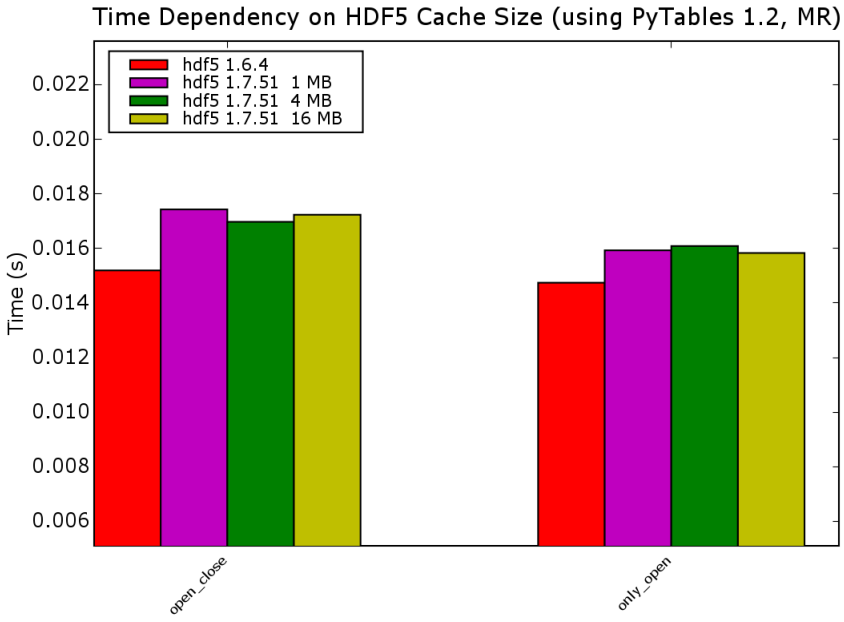

Figure 8: Opening times for PyTables 1.2

future HDF5 1.8.x series, it is interesting to check that PyTables 1.2 seems to not be much affected by the future version of HDF5. In addition, the fact that HDF5 1.7.51 is a preliminary version of the future HDF5 1.8 might allow the HDF group to enhance the performance in the meanwhile.

In order to see exactly how much the opening times are in PyTables 1.2, and if they depend on the HDF5 version, in figure 8 is shown a zoom of just these times. There, it can be assessed how fast the opening is now, and that there is no much difference in using HDF5 1.6.4 or HDF5 1.7.51 (whatever the size of metadata cache was chosen).

2. Memory usage effects

Let's see now what happens with HDF5 1.7.51 and memory consumption. In Figure 9, we can see how memory usage is favourable (up to a 25%) in all cases to HDF5 1.7.51. This is actually good news and perhaps the most important goal for the new metadata cache in HDF5 1.8.x series.

We can also see that for the *full_browse* and *full_browse_attrs*, the difference in terms of memory consumption between PyTables 1.1.1 and 1.2 is quite important, and that the combination of PyTables 1.2 and HDF 1.7.51 is, by far, the best in that regard (the memory usage is reduced almost a 2x compared with PyTables 1.1.1 and HDF5 1.6.4).

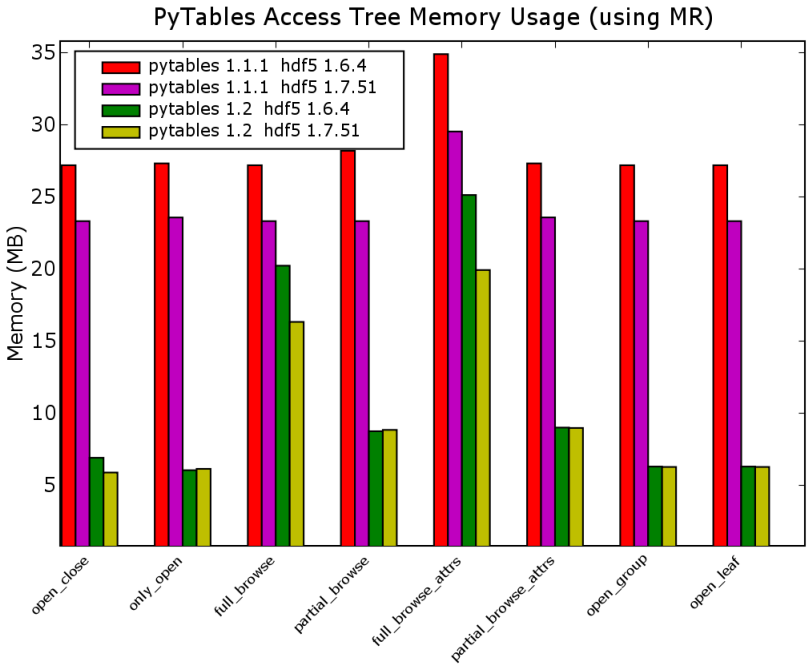

Figure 9: Memory usage in HDF5 1.6.4 vs 1.7.51

3. *Effects of changing the size of cache in HDF5 1.7.51*

As HDF5 1.7.51 has a configurable size for metadata cache, we want also to check the effects of changing it. The Figure 10 shows the memory consumption as a function of the HDF5 1.7.51 cache size. On it, one can observe that, using the default size of 1 MB, HDF5 1.7.51 represents a savings of 15% more or less on every use case, which is a good figure. However, making the cache bigger (4 MB and 16 MB) makes the memory consumption to grow to values comparable to HDF5 1.6.4. This can be seen as normal, because a bigger cache size should imply more memory consumption.

If we redo the same graph but using PyTables 1.2 instead of 1.1.1, we get the Figure 11. Here, we can observe the same behaviour as we have seen above, i.e. that increasing the cache size in HDF5 library effectively increase the memory needs.

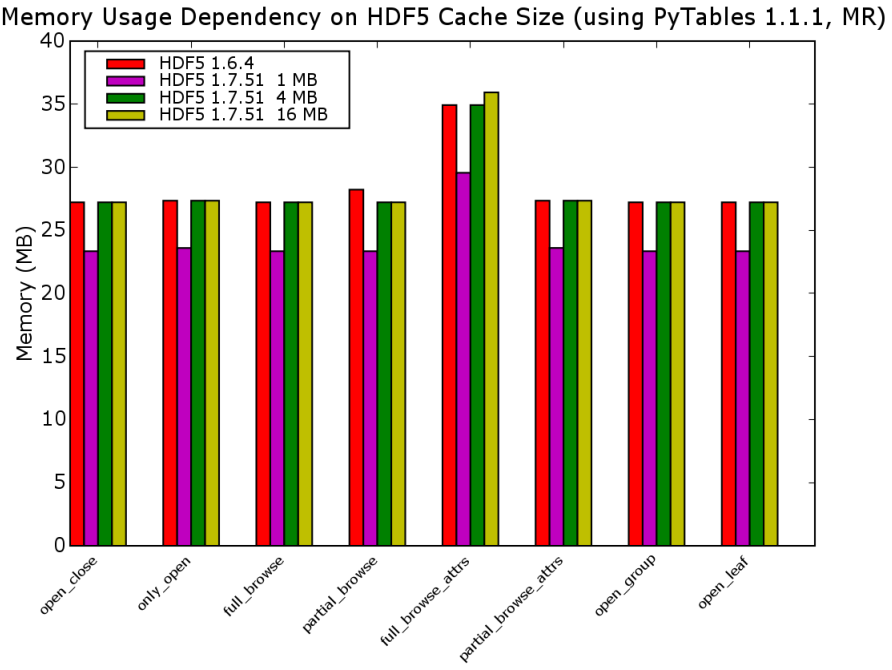

Figure 10: Memory usage in PyTables 1.1.1 and different cache sizes for HDF5 1.7.51

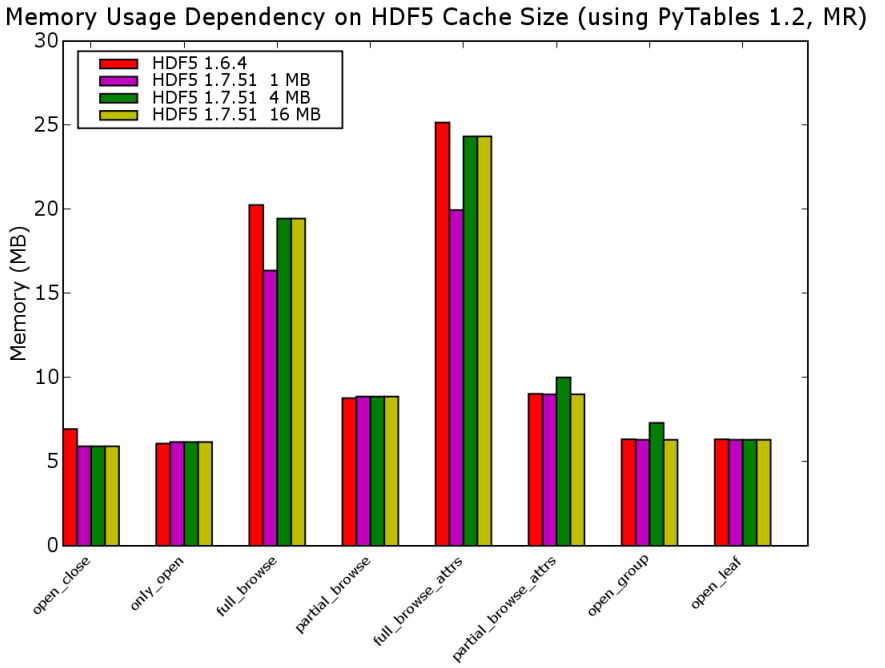

Figure 11: Memory usage in PyTables 1.2 and different cache sizes for HDF5 1.7.51

IV. CONCLUSIONS

From the described experiments several conclusions can be stated:

1. The new LRU cache in PyTables 1.2 considerably reduces (typically, more than a factor 100x compared with Pytables 1.1.1 and more than 200x compared with PyTables 1.1) the *opening* time for files with a large number of objects. At typically less than 1 tenth of second of opening latency, PyTables 1.2 lets the users to be able to interactively work with files with an arbitrarily large number of groups and datasets on it.

2. However, when accessing all the nodes in the object tree, the new LRU cache algorithm actually slows down the access times (up to a 2x in some situations, like the *full_browse_attrs* test). Some profiling has been carried out and determined that the slowdown is due to the addition of the LRU cache. Some solutions has been proposed to improve this access to metadata in these scenarios (see FUTURE WORK section below).

3. The HDF5 1.7.51 version has been tested as well as HDF5 1.6.4. The times for opening using 1.7.51 seem to degrade slightly when used with PyTables 1.2 (up to a 5%), and more if used with PyTables 1.1.1 (up to a 30%). However, this performance can be improved by when HDF5 1.8 would be released. In any case, the issue does not seem to be grave for the purposes of this study.

4. Regarding memory consumptions, the HDF5 1.7.51 library seems to work much better than 1.6.4, specially when used with PyTables 1.2 were it can reduce the memory needs up to a 2x. The combination of PyTables 1.2 and the forthcoming HDF5 1.8 will allow to deal with files with really large amounts of nodes on a very efficient way.

V. FUTURE WORK

Based on the experiments presented here, there are a couple of paths of work that can be followed in the future:

1. In order to improve times in scenarios where one need to access to the complete object tree, we suggest to push the **lrucache** module down to Pyrex.

2. It is apparent that HDF5 1.7.51 represents a nice achievement in terms of memory consumption. However, it should be tuned a little bit in order to improve times (at least until reaching HDF5 1.6.4 performance), specially when reading of attributes or opening of nodes.