# PyTables

## Processing And Analyzing Very Large Amounts Of Data In Python

Francesc Alted
falted@openlc.org

EuroPython 2003, 25 June, 2003, Charleroi, Belgium

# Outline

- What is PyTables and why it exists?
- Interactive demonstration
- Some benchmarks
- Final remarks

# Motivation

- Many applications need to save and read very large amounts of data ==> processing it is a real challenge!

- Computers are powerful enough to deal with very large data sets. But, the question is: can people handle such data sets?

- Requirements:
  - Analysis is an iterative process: interactivity
  - Re-reading many times the data: efficency
  - Good framework to endow the data a structure

- PyTables is a Python package designed with these requirements in mind!

# What does PyTables offer?

- Interactivity
  - The user can take immediate action based on previous feedback
  - This greatly accelerates the process of data mining

- Efficiency
  - Improves your productivity
  - Very important when interactivity is an issue

- Hierarchical structure
  - It allows you to break your data into smaller, related chunks
  - It offers you an intuitive way to categorize data
  - Datasets become objects that can be easily manipulated

# Machinery behind PyTables

PyTables relies on powerful software to achieve its goals:

- Python -- everyone here knows that (2.2 version needed because generators are heavily used)
- HDF5 -- general purpose library and file format for storing scientific data
- numarray -- next generation of the well-known Numerical Python package
- Pyrex -- tool to make Python extensions with a Python-like syntax

# What is HDF5?

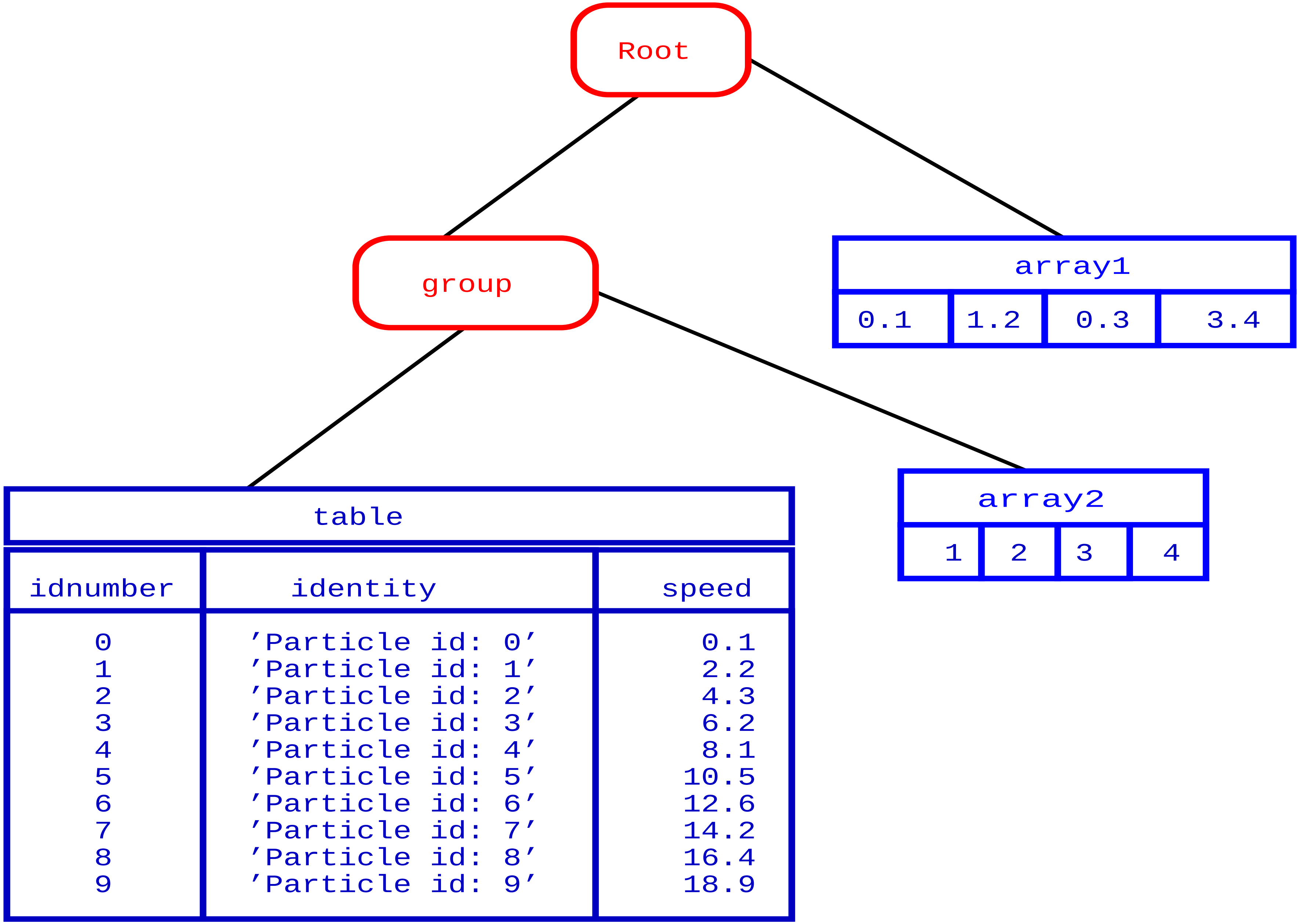
It is a general purpose library and file format for storing scientific data in a hierarchical manner. It is developed and maintained at NCSA.

- Can store two primary objects: datasets and groups
  - Dataset: multidimensional array of data elements
  - Group: Structure for organizing objects in the HDF5 file
- Very flexible and well tested in scientific environments
- Being already used in: Meteorology, Oceanography, Astronomy, Astrophysics, Numerical simulation and many other applications

# PyTables highlights

- General Python library to deal with large amounts of data
- Support of Numerical Python and numarray objects
- Appendable tables
- Can read generic HDF5 files
- Transparent data compression support
- Support of files bigger than 2 GB (unlimited data size in practice)
- Architecture-independent (is aware of big/little endian issues)

# A first example



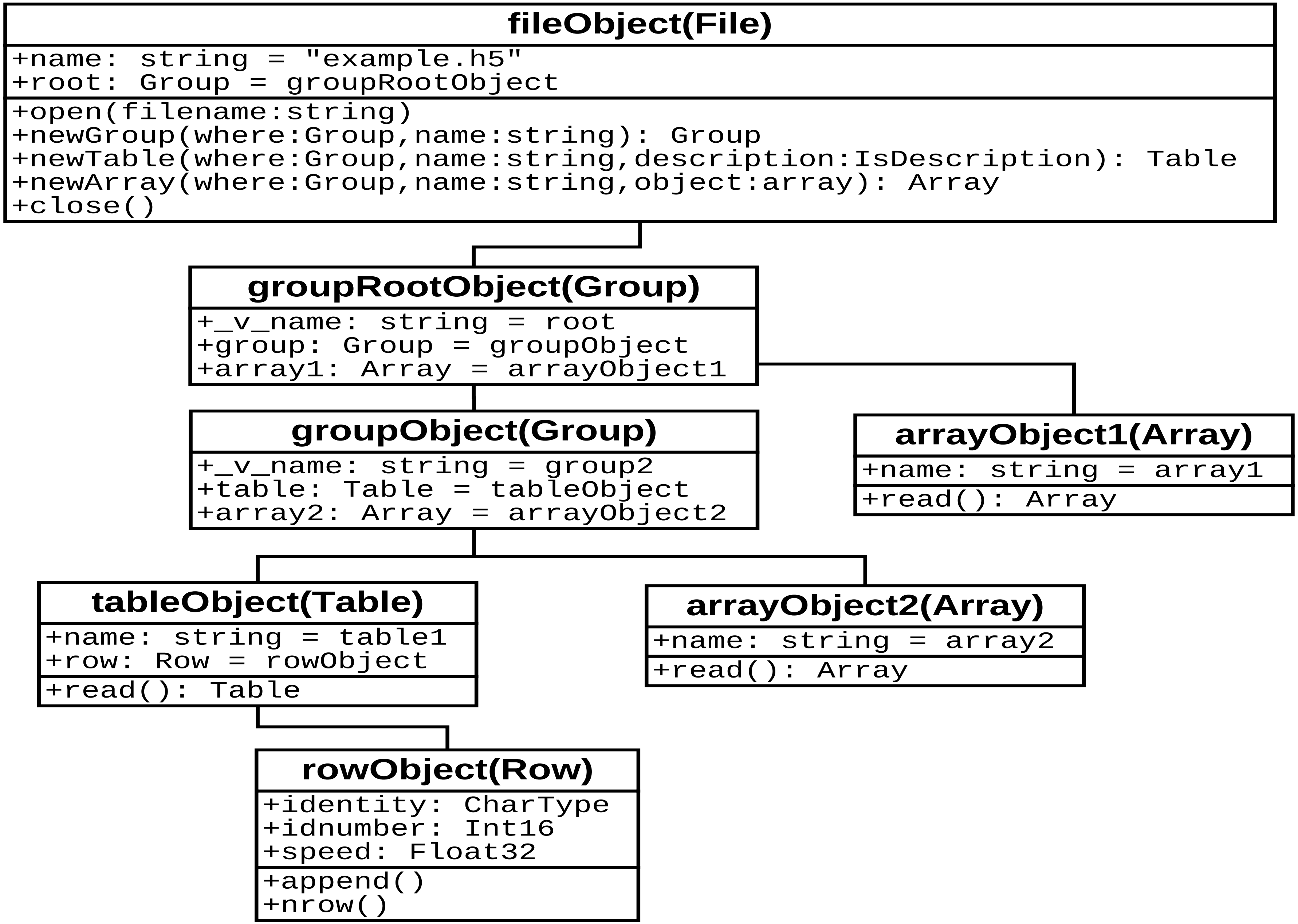
# The PyTables code

```
from tables import *

class Particle(IsDescription):
    identity = Col("CharType", 16, " ", pos = 0)  # character String
    speed = Col("Float32", 1, pos = 2)  # single-precision
    idnumber = Col("Int16", 1, pos = 1)  # short integer

fileh = openFile("example.h5", mode = "w")
fileh.createArray(fileh.root, "array1", [.1,.2,.3,.4], "Floats")
group = fileh.createGroup(fileh.root, "group")
fileh.createArray(group, "array2", [1,2,3,4], "Int array")
table = fileh.createTable(group, "table", Particle, "3 fields")
row = table.row
for i in xrange(10):
    row['identity']  = 'Particle id: %3d' % (i)
    row['idnumber'] = i
    row['speed']  = i * 2.
    row.append()

fileh.close()
```

# First example output

```
$ h5ls -rd example.h5
/array1                 Dataset {4}
    Data:
        (0) 0.1, 0.2, 0.3, 0.4
/group                  Group
/group/array2               Dataset {4}
    Data:
        (0) 1, 2, 3, 4
/group/table              Dataset {10/Inf}
    Data:
        (0) {0, "Particle id:   0", 0}, {1, "Particle id:   1", 2},
        (2) {2, "Particle id:   2", 4}, {3, "Particle id:   3", 6},
        (4) {4, "Particle id:   4", 8}, {5, "Particle id:   5", 10},
        (6) {6, "Particle id:   6", 12}, {7, "Particle id:  7", 14},
        (8) {8, "Particle id:   8", 16}, {9, "Particle id:  9", 18}
```

# The object tree



# How fast is fast?

- Several benchmarks have been conducted in order to analyze if PyTables is competitive with existing tools to save data persistently.
- Comparisons has been made with cPickle, struct, and SQLite (a relational database).
- The benchmarks tested writing and selecting table data that fulfill a series of conditions.
- The effect of transparent data compression has also been analyzed.

# The row descriptions

Two different row sizes of different lengths has been choosed:

- Small Size (16 Bytes)

```
class Small(IsDescription):
    var1 = Col("CharType", 4, "")
    var2 = Col("Int32", 1, 0)
    var3 = Col("Float64", 1, 0)
```

- Medium Size (56 Bytes)

```
class Medium(IsDescription):
    name       = Col("CharType", 16, "")
    float1     = Col("Float64", 2, NA.arange(2))
    ADCcount   = Col("Int32", 1, 0)
    grid_i     = Col("Int32", 1, 0)
    grid_j     = Col("Int32", 1, 0)
    pressure   = Col("Float32", 1, 0)
    energy     = Col("Float64", 1, 0)
```

# The selection mechanism

- PyTables:
  - e = [ row['var1'] for row in table.iterrows() if row['var2'] < 20 ]

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

- cPickle:
  - while rec:

```
while rec:
    record = cPickle.loads(rec[1])
    if record['var2'] < 20:
        e.append(record['var1'])
```

- struct:
  - while rec:

```
while rec:
    record = struct.unpack(isrec._v_fmt, rec[1])
    if record[1] < 20:
        e.append(record[0])
```

- SQLite:
  - cursor.execute("select var1 from table where var2 < 20")

Note: cPickle and struct tests use a RECNO bsddb3 database in order to emulate records efficently.

# Benchmark platform description

- System 1
  - Laptop with Intel P4 @ 2 GHz
  - 256 MB RAM
  - Disk IDE @ 4200 RPM
- System 2
  - Workstation with AMD XP @ 1.8 GHz
  - 1024 MB RAM
  - Disk IDE @ 7200 RPM

- PyTables pre-0.6
- HDF5 1.4.5-post2
- numarray pre-0.6
- SQLite 2.8.3
- PySQLite 0.4.3

# Comparing cPikle and struct with PyTables

| Package | Record length | Krows/s | | MB/s | | total Krows | file size | memory used | %CPU | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | write | read | write | read | | (MB) | (MB) | write | read |
| cPickle | small | 23.0 | 4.3 | 0.65 | 0.12 | 30 | 2.3 | 6.0 | 100 | 100 |
| cPickle | small | 22.0 | 4.3 | 0.60 | 0.12 | 300 | 24 | 7.0 | 100 | 100 |
| cPickle | medium | 12.3 | 2.0 | 0.68 | 0.11 | 30 | 5.8 | 6.2 | 100 | 100 |
| cPickle | medium | 8.8 | 2.0 | 0.44 | 0.11 | 300 | 61 | 6.2 | 100 | 100 |
| struct | small | 61 | 71 | 1.6 | 1.9 | 30 | 1.0 | 5.0 | 100 | 100 |
| struct | small | 56 | 65 | 1.5 | 1.8 | 300 | 10 | 5.8 | 100 | 100 |
| struct | medium | 51 | 52 | 2.7 | 2.8 | 30 | 1.8 | 5.8 | 100 | 100 |
| struct | medium | 18 | 50 | 1.0 | 2.7 | 300 | 18 | 6.2 | 100 | 100 |
| PyTables | small | 434 | 469 | 6.8 | 7.3 | 30 | 0.49 | 6.5 | 100 | 100 |
| PyTables | small (c) | 326 | 435 | 5.1 | 6.8 | 30 | 0.12 | 6.5 | 100 | 100 |
| PyTables | small | 663 | 728 | 10.4 | 11.4 | 300 | 4.7 | 7.0 | 99 | 100 |
| PyTables | medium | 194 | 340 | 10.6 | 18.6 | 30 | 1.7 | 7.2 | 100 | 100 |
| PyTables | medium (c) | 142 | 306 | 7.8 | 16.6 | 30 | 0.3 | 7.2 | 100 | 100 |
| PyTables | medium | 274 | 589 | 14.8 | 32.2 | 300 | 16.0 | 9.0 | 100 | 100 |

**Table 1:** Comparing PyTables performance with cPickle and struct serializer modules in Standard Library. (c) means that a compression is used.

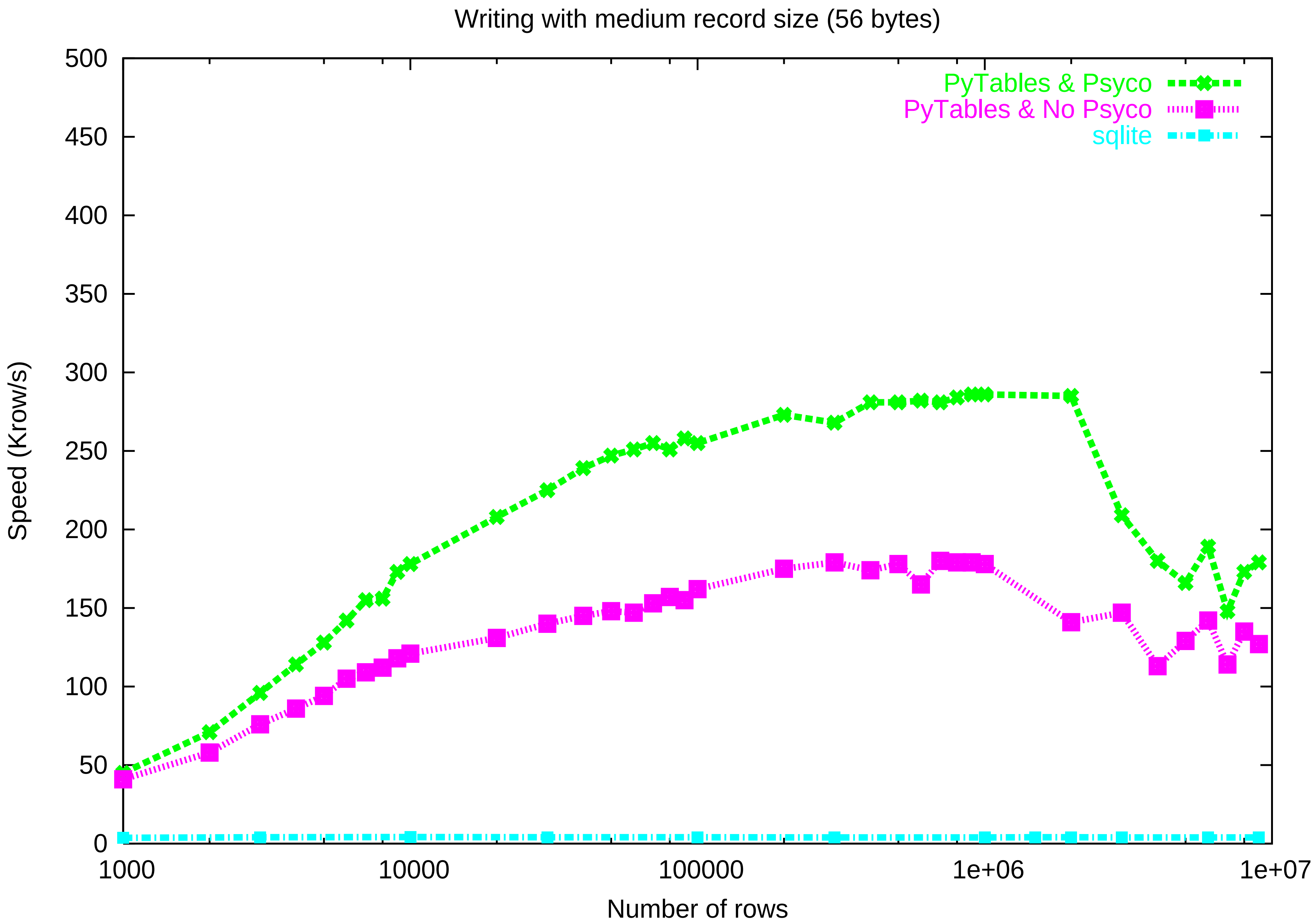
# Conclusions from first benchmark (cPickle & struct)

- Writing
  - Between 20 and 30 times faster than cPickle + bsddb3
  - Between 3 and 10 times faster than struct + bsddb3

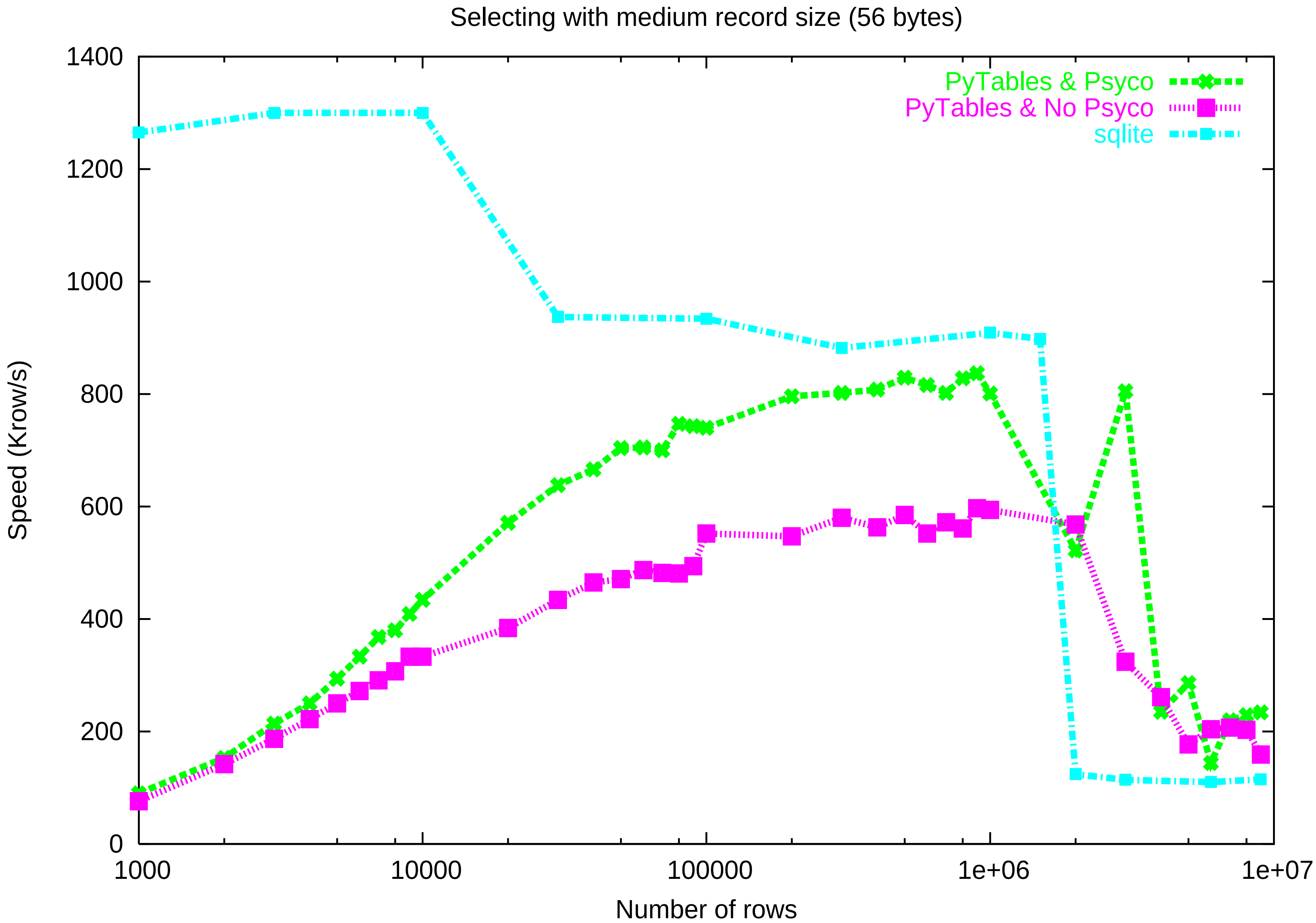
- Reading
  - Around 100 times faster than cPickle + bsddb3
  - Around 10 times faster than struct + bsddb3

PyTables is far superior to cPickle and struct for any amount of data

# Comparing SQLite with PyTables (writing)



# Comparing SQLite with PyTables (selecting)



# PyTables vs SQLite (conclusions)

Writing

- PyTables is around 35 times faster than SQLite
- Note: SQLite runs in asynchronous mode (i.e. the fastest)

Reading

- In-core selects (i.e. file size fits in cache memory)
  - PyTables achieves between 60% and 90% of SQLite speed
- Out-of-core selects (i.e. file size do not fit in cache memory)
  - PyTables outperforms SQLite between a 30% and a 100%

PyTables beats SQLite when dealing with large amounts of data!
(while being close to it for smaller sizes)

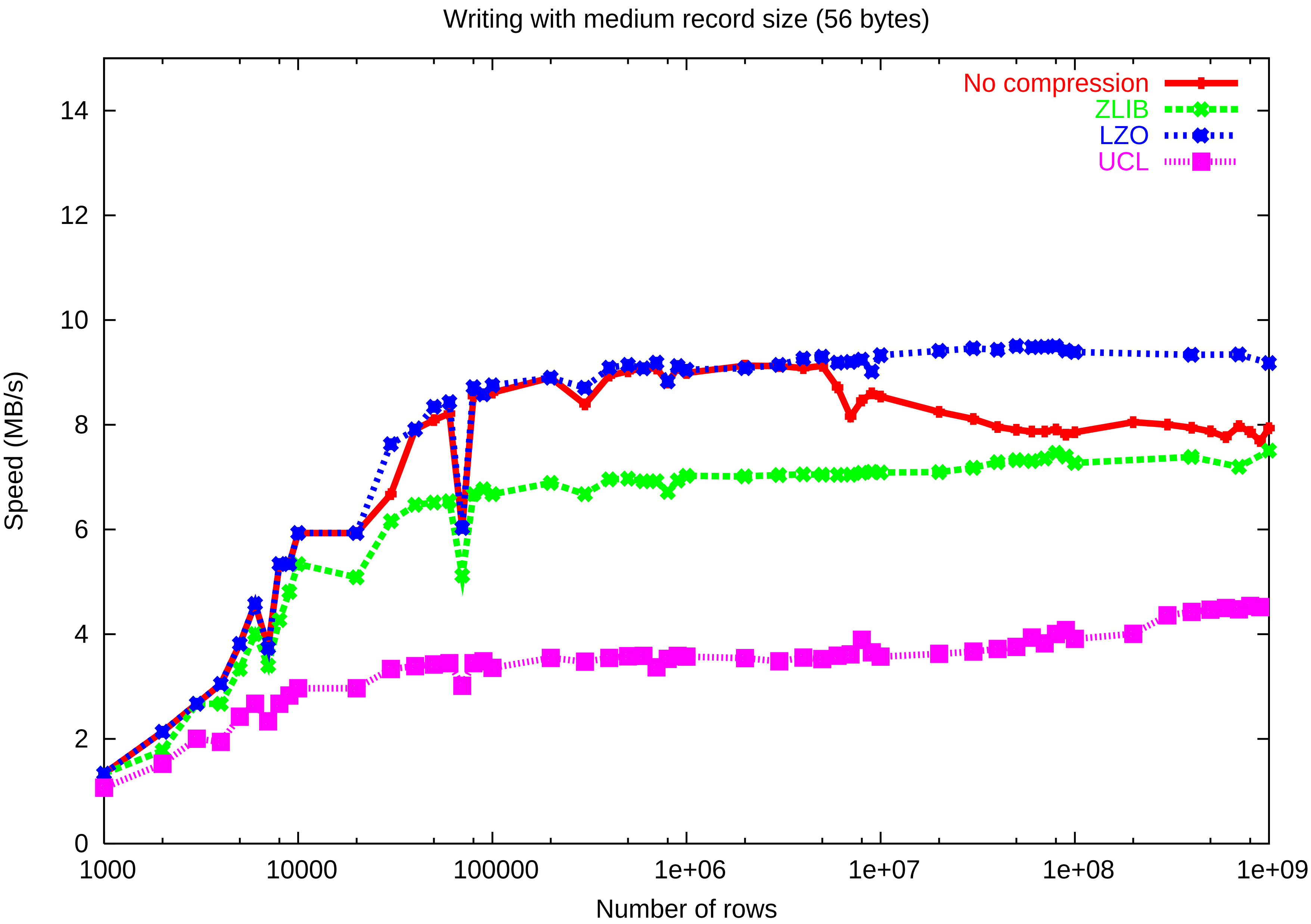
# Wave of the future: Compression

- Compression alleviates disk limitations in exchange of consuming more CPU

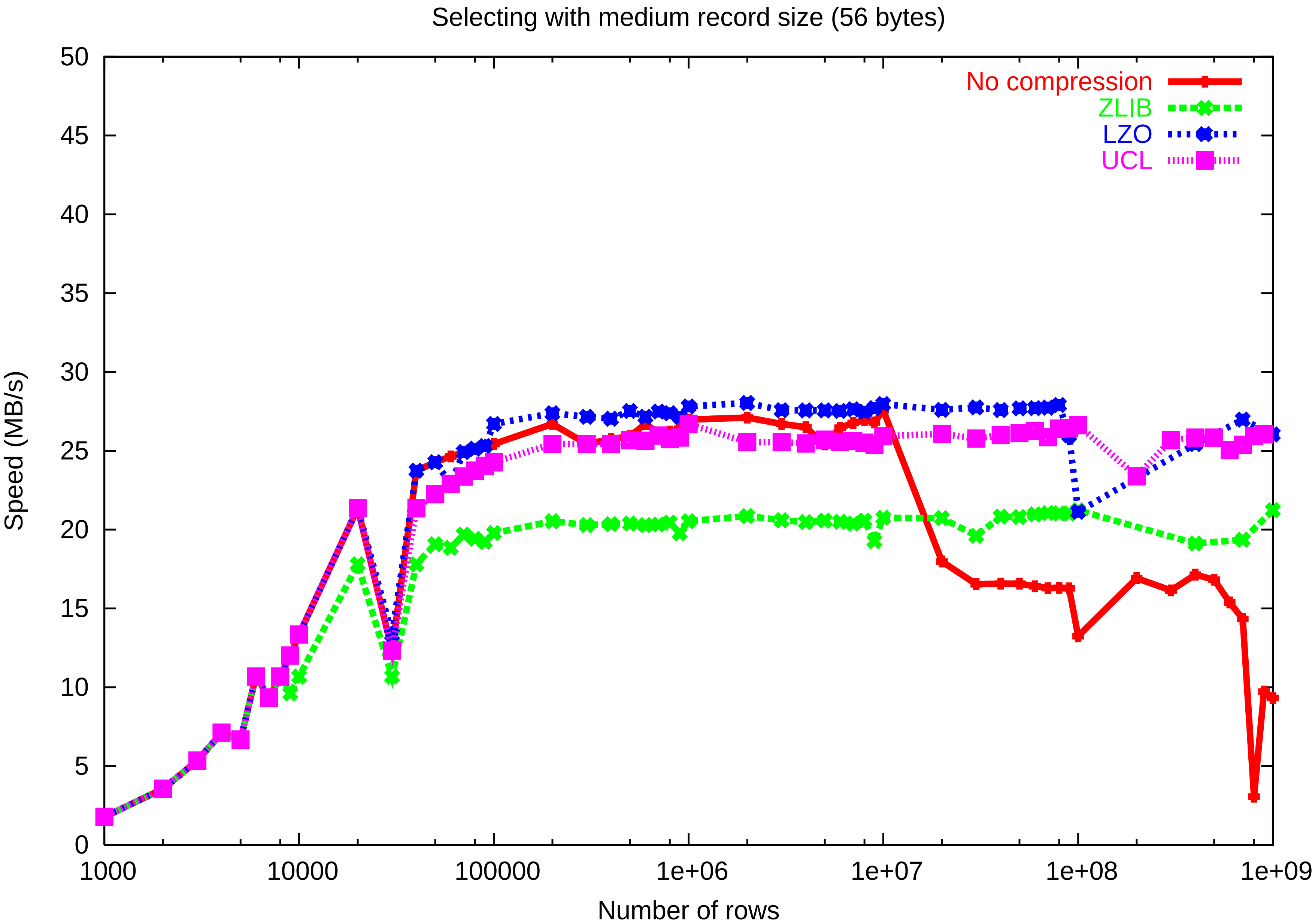
- CPU speed grows much faster than disk speed and capacities:
  - CPU speed grows a 60% / year
  - Disk capacity grows a 30% / year
  - Disk bandwith only grows a 20% / year

- Compression will increasingly help to speed-up the I/O process as well as to expand the capacity capabilities of disks

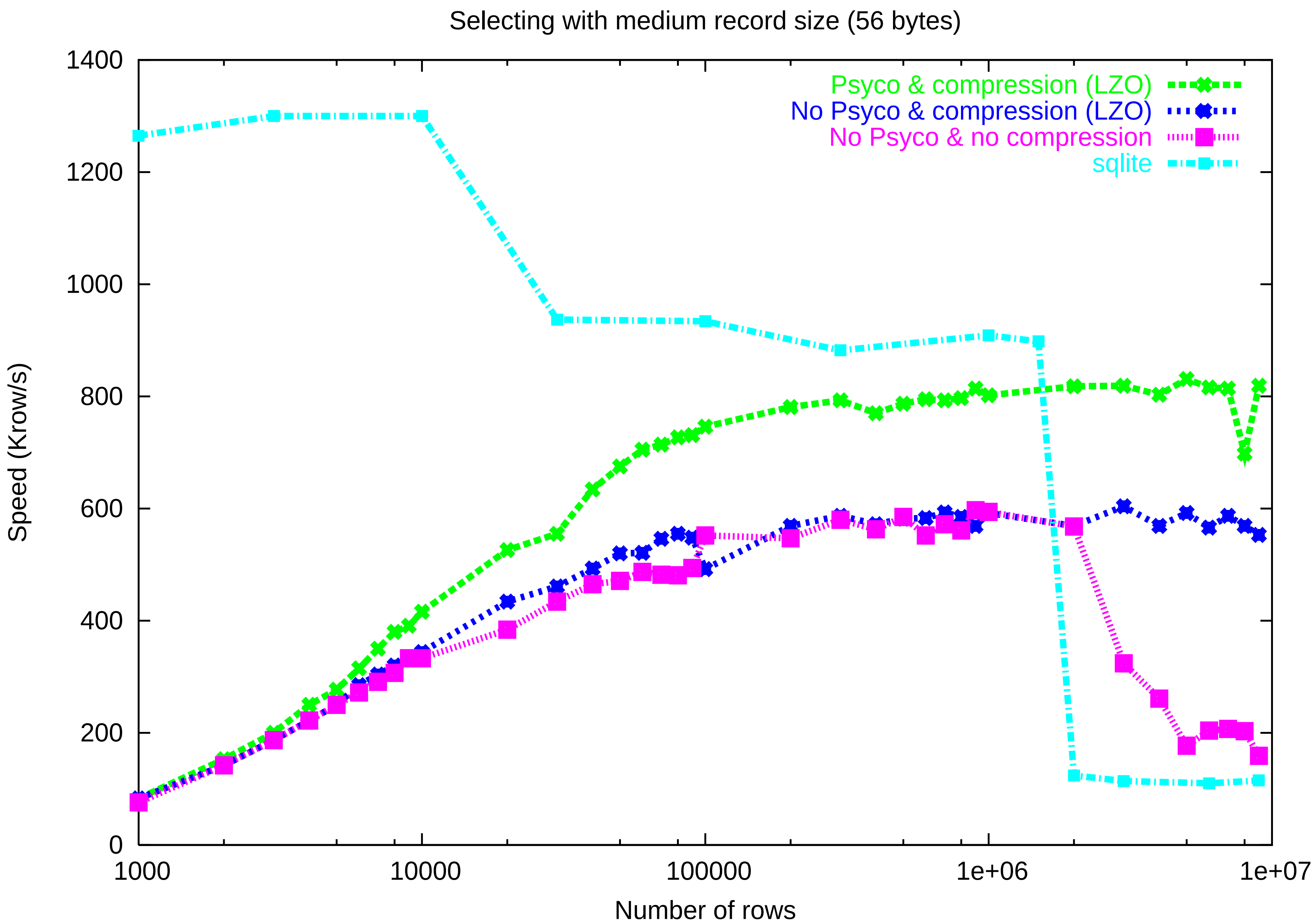
# Compression benchmarks (writing)



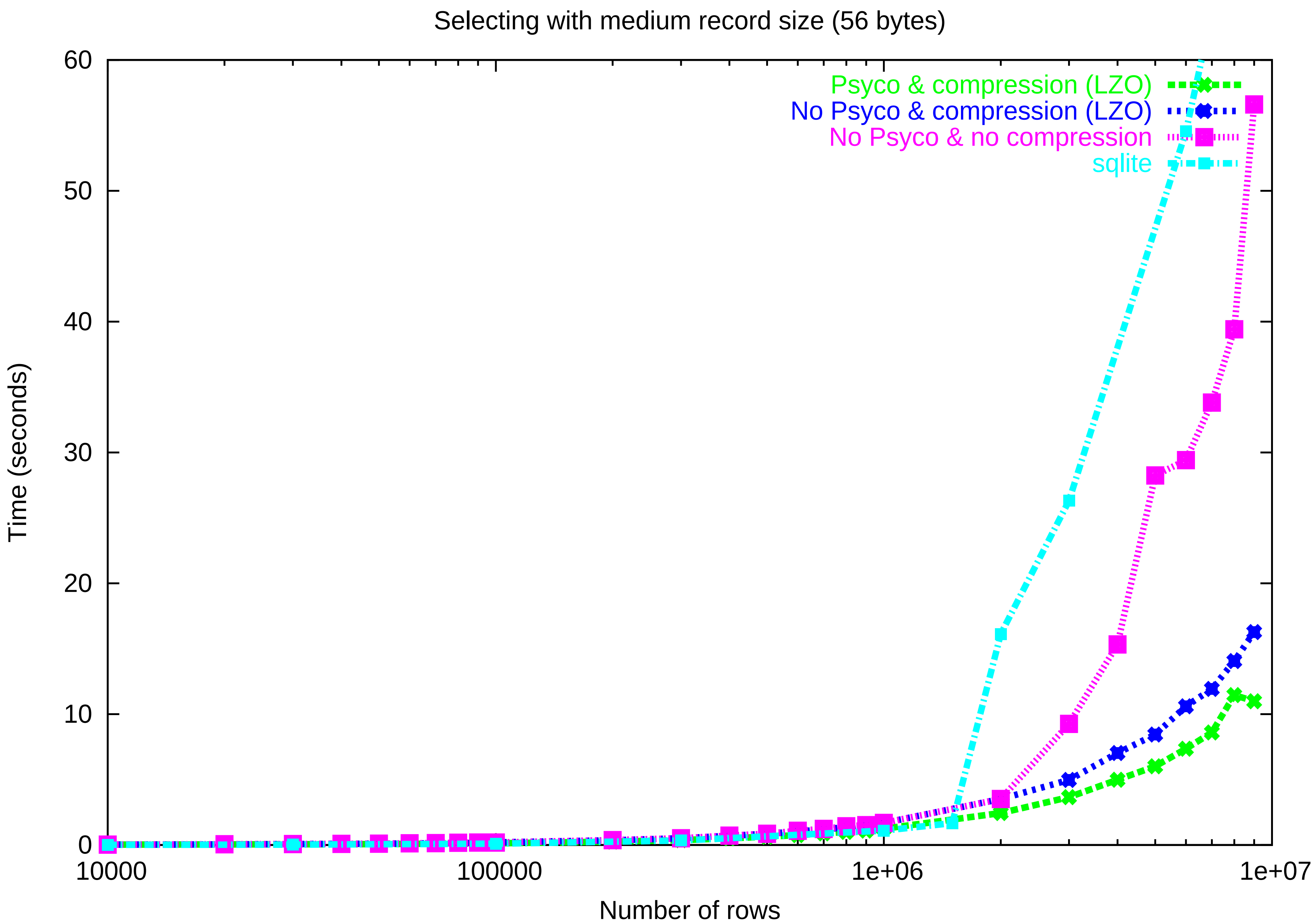
# Compression benchmarks (reading)



# Comparison with SQLite revisited (I)



# Comparison with SQLite revisited (II)



# Compression benchmarks (conclusions)

- Compression improves the reading speed by a factor between 1.5 and 2 (that depends on the compressor choosed and the dataset size).

- When writing, only a small fraction of the original speed is lost (except with the LZO compressor, which is as efficient as the no-compression case).

- When compression is used together with Psyco, PyTables can be up to 8 times faster than SQLite for the out-of-core case.

- Compression also expands the data size range where the filesystem can make use of the system memory to cache the file.

# Current PyTables limitations and plans for future

- One can not delete a single row on a table. You need to rewrite the whole table except the row you want to remove. This will hopefully be solved when the next release of HDF5 (1.6) appears.

- Elements in columns can not have more than one dimension. This should be solved when numarray 0.6 appears (it will have support for multidimensional recarrays cells).

- Object or row elements can not be related to other elements

- More filters have to be added to import data from other data sources, such as NetCDF, ASCII, CSV, etc. files.

# PyTables uses

Situations were data has to be acquired once and read multiple

- Scientific Applications
  - Meteorology
  - Astronomy
  - Experimental Physics
  - Medicine (Physiological sensors)
  - ...
- Data acquisition from IT applications
  - Tracing data from routers
  - System monitoring
  - Security (Firewalls, IDS, ...)
  - ...

# Final remarks

- PyTables allows you to process your data interactively and quickly.

- If you have large amounts of data, an interpreted language like Python is more than enough to get maximum performance: PyTables is only limited by disk I/O speed.

- PyTables has been designed to excel in retrieving and selecting data very fast, but is also stunningly fast when writing (I didn’t expect this result - a welcome surprise).

# PyTables is for real work!

- More than 200 tests units are now incorporated. More will be added and quality will only improve as PyTables evolves.
- PyTables is already in beta and its API is mostly stable.
- It comes with complete documentation both in doc strings format as well as in a high quality 50 pages user's manual in PDF and HTML formats.
- Download the last version (0.5.1, released on May 13th, and 0.6 is close to publication) and use it for free from:

http://pytables.sourceforge.net