Data Oriented Programming

or:

How I Learned to Stop Worrying about CPU Speed and Love Memory Access

Francesc Alted
Freelance Trainer And Developer

PyData Conference, Berlin
July 26, 2014

Overview

- Motivation
- The Data Access Issue
  - Why Modern CPUs Are Starving
  - Why Caches?
  - Techniques For Fighting Data Starvation
- Optimal Containers for Big Data

About Me

- I am the creator of tools like PyTables, Blosc, bcolz, and a long-term maintainer of Numexpr
- I am an experienced **developer and trainer** in:
  - Python (almost 15 years of experience)
  - High Performance Computing and Storage
- Also available for **consulting**

Motivation

CPU Speed Is No Longer *The Holy Grail*

Supose that we want to compute the next polynomial:

$$0.25x^3 + 0.75x^2 + 1.5x - 2$$

in the range [-1, 1] with a step size of $2*10^{-7}$ in the $\times$ axis

...and want to do that as FAST as possible...

Using NumPy

```
import numpy as np

N = 10*1000*1000

x = np.linspace(-1, 1, N)

y = .25*x**3 + .75*x**2 - 1.5*x - 2
```

That takes around 1.60 sec on some machine (Intel Xeon E5520 @ 2.3 GHz). How to make it faster?

'Quick & Dirty' Approach: Parallelize

- Computing a polynomial is "embarrassingly" parallelizable: just divide the domain to compute in N chunks and evaluate the expression for each chunk.
- This can be easily implemented in Python by using the multiprocessing module (so as to bypass the GIL).
- Using 2 cores, the 1.60 sec is reduced down to 1.18 sec, which is a 1.35x improvement. Not bad.
- We are done! Or perhaps not?

A Better Approach: Optimize

The NumPy expression:

(I) $\mathbf{0.25x^3 + 0.75x^2 + 1.5x - 2}$

can be rewritten as:

(II) $\mathbf{((0.25x + 0.75)x + 1.5)x - 2}$

- Exec time goes from 1.60 sec to 0.30 sec
- Much faster (4x) than using two processors with the multiprocessing approach (1.18 sec).

First Take Away Message

- **Do not blindly try to parallelize right away: Optimizing normally gives better results**

  And a serial codebase is normally much easier to code and debug!

Use numexpr

Numexpr is a JIT compiler, based on NumPy, that optimizes the evaluation of complex expressions. Usage is simple:

```
import numpy as np
import numexpr as ne
N = 10*1000*1000
x = np.linspace(-1, 1, N)
y = ne.evaluate(".25*x**3 + .75*x**2 - 1.5*x - 2")
```

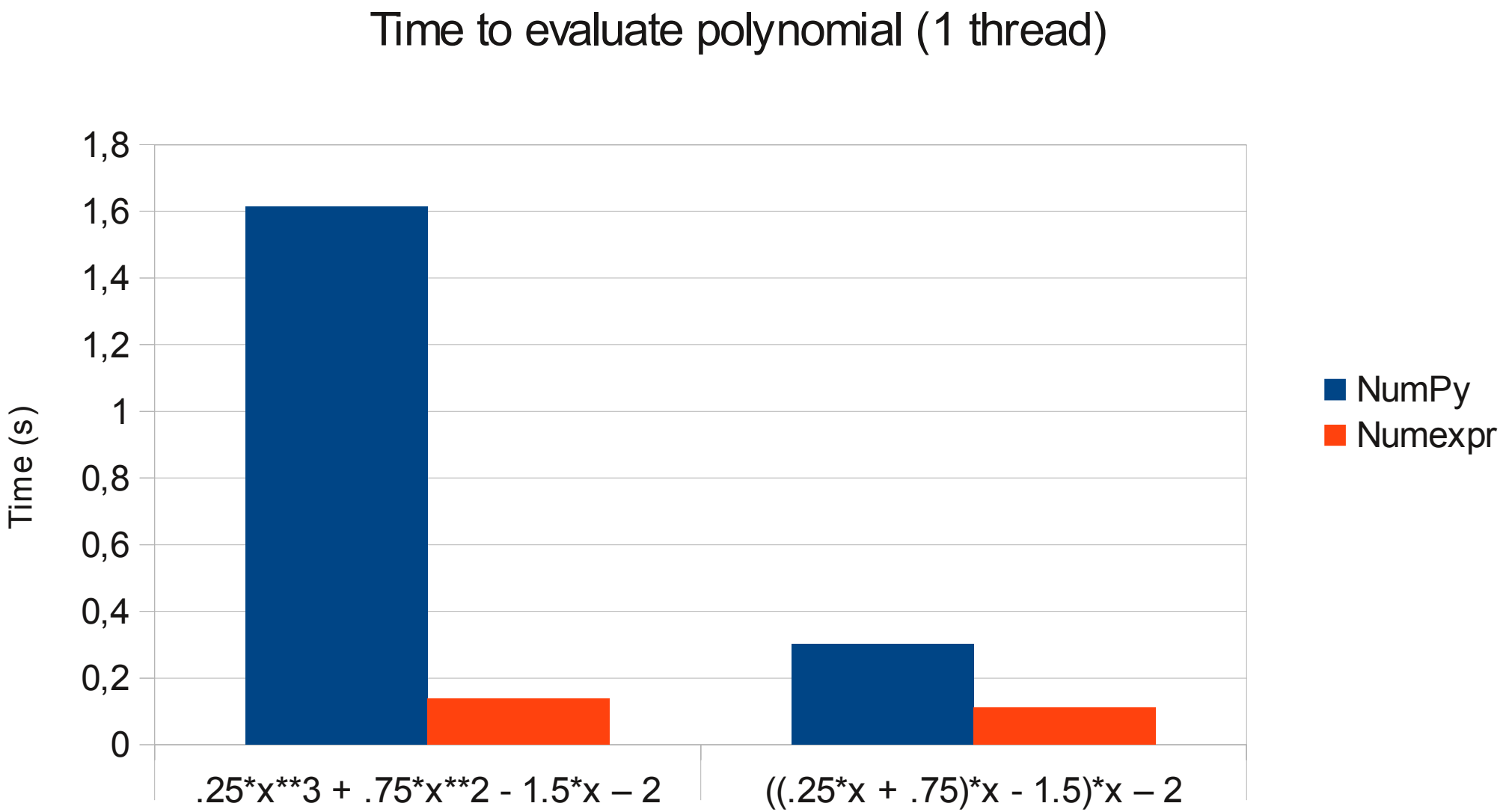
This takes 0.14 sec to complete (11x faster than the original NumPy: 1.60 sec)

Fine Tuning numexpr

Numexpr is also sensible to computer-friendly expressions like:

(II) **((0.25x + 0.75)x + 1.5)x - 2**

- This takes 0.11 sec (3x faster than NumPy)
- 0.14 sec were needed for the original expression, that's a 25% faster


Time to evaluate polynomial (1 thread)
Time (s)
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
.25*x**3 + .75*x**2 - 1.5*x – 2
((.25*x + .75)*x - 1.5)*x – 2
NumPy
Numexpr

Power Expansion

Numexpr expands expression:

```
0.25*x**3 + 0.75*x**2 + 1.5*x - 2
```

to:

```
0.25*x*x*x + 0.75*x*x + 1.5*x - 2
```

so, no need to use the expensive pow()

One Remaining Question

Why numexpr can execute this expression:

((0.25x + 0.75)x + 1.5)x - 2

3x faster, even using a single core?

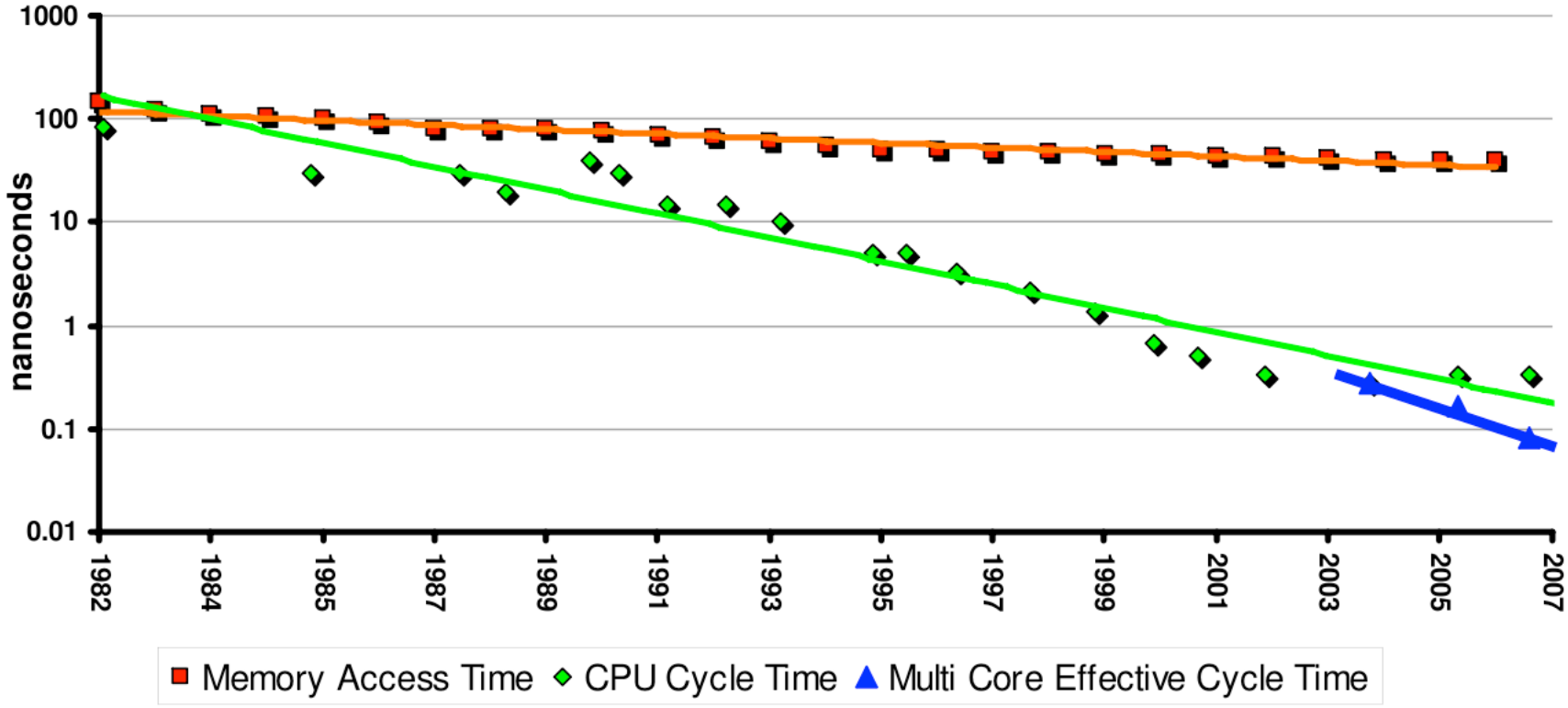
Short answer: making a more efficient use of the memory resource

The Starving CPU Problem

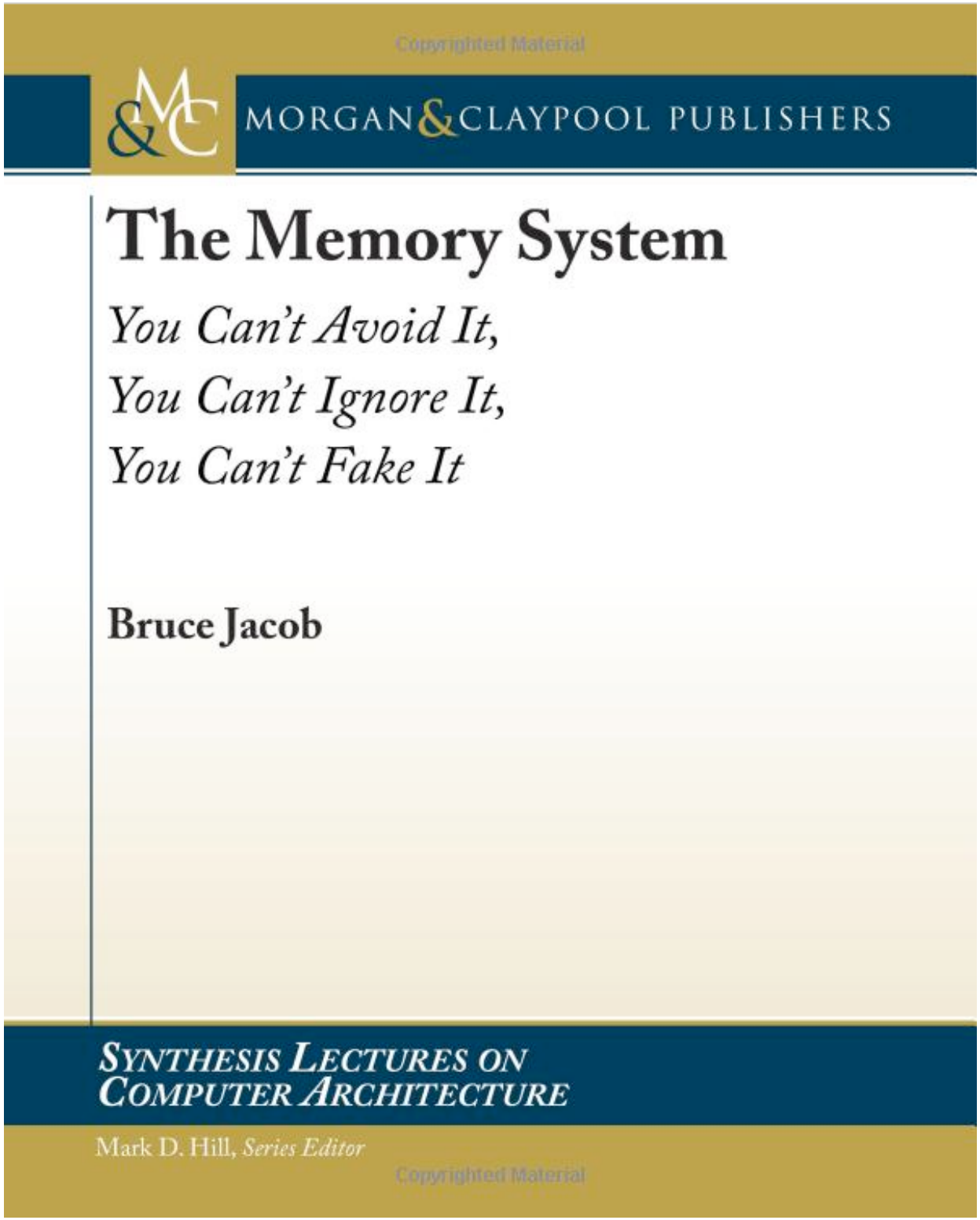
The Starving CPU Problem

- Current CPUs typically stay bored, doing nothing most of the time
- Why so?
- Because they are basically **waiting for data**

Memory Access Time vs CPU Cycle Time



Book in 2009


Copyrighted Material
MORGAN & CLAYPOOL PUBLISHERS
The Memory System
You Can't Avoid It,
You Can't Ignore It,
You Can't Fake It
Bruce Jacob
SYNTHESIS LECTURES ON
COMPUTER ARCHITECTURE
Mark D. Hill, Series Editor
Copyrighted Material

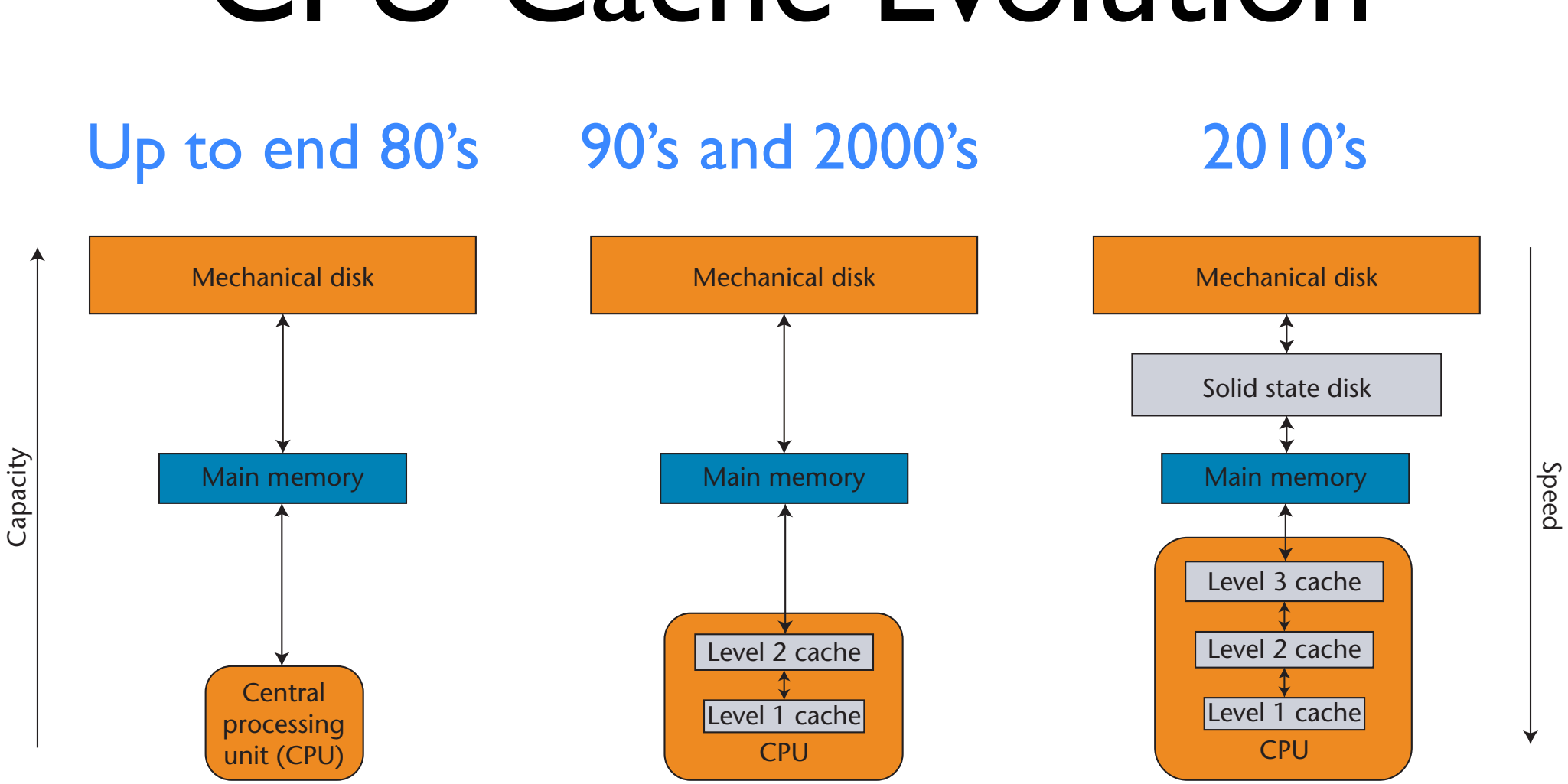
The Status of CPU Starvation in 2014

- Memory latency is much slower (between 100x and 500x) than processors.
- Memory bandwidth is improving at a better rate than memory latency, but it is also lagging behind processors (between 30x and 100x).

CPU Caches to the Rescue

- CPU cache latency and throughput are much better than memory
- However: the faster they run the smaller they must be

CPU Cache Evolution



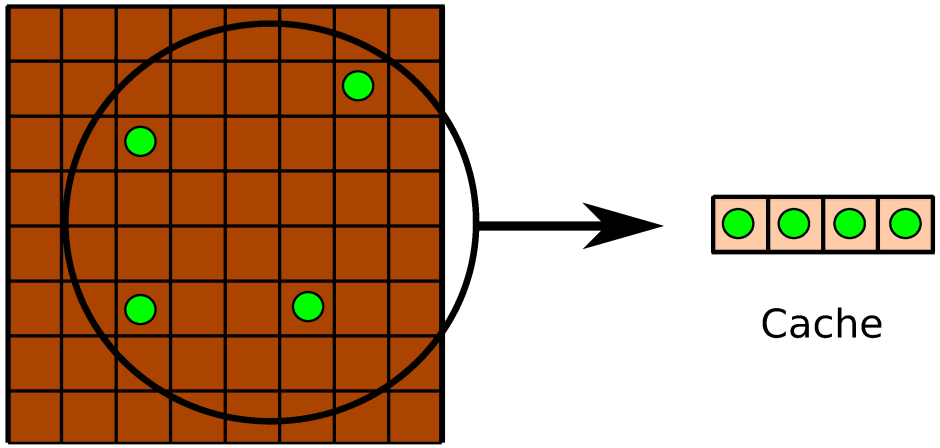
When CPU Caches Are Effective?

Mainly in a couple of scenarios:

- Time locality: when the dataset is reused
- Spatial locality: when the dataset is accessed sequentially

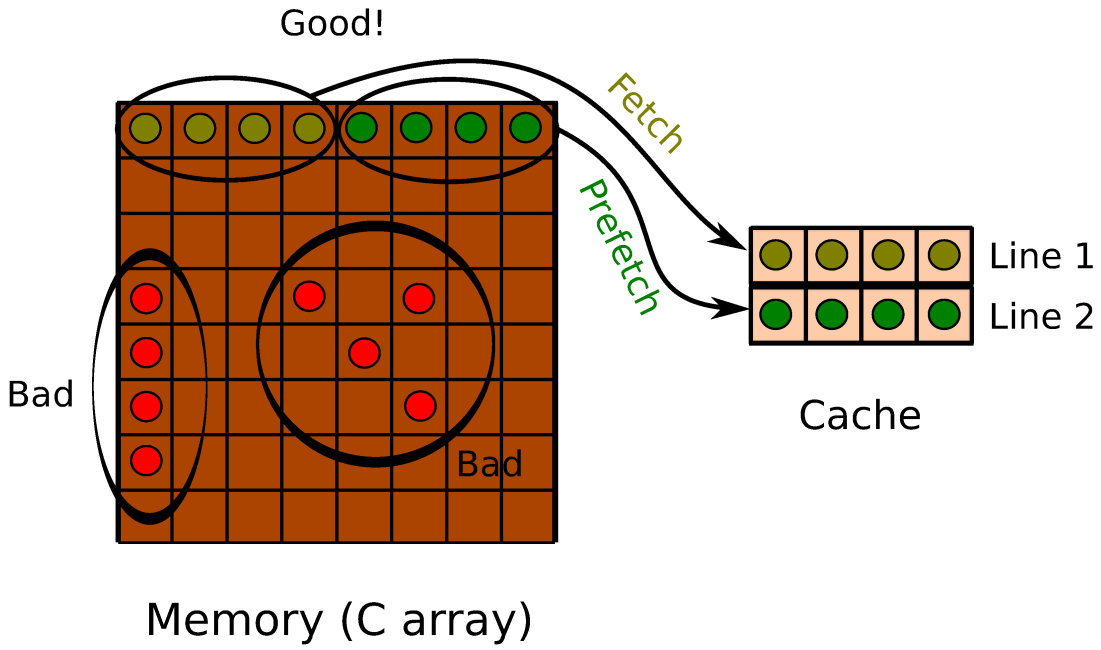
Time Locality

Parts of the dataset are reused



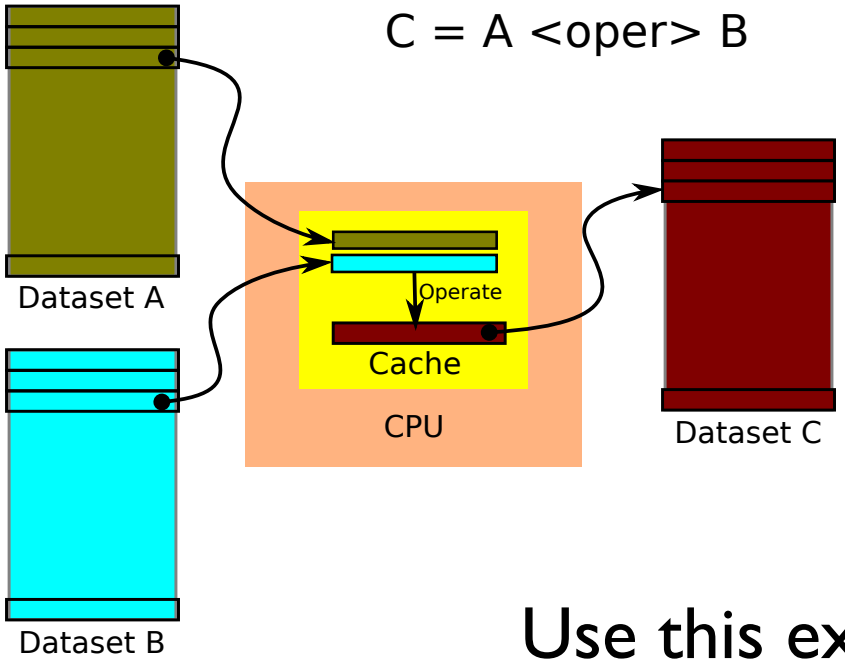
Spatial Locality

Dataset is accessed sequentially



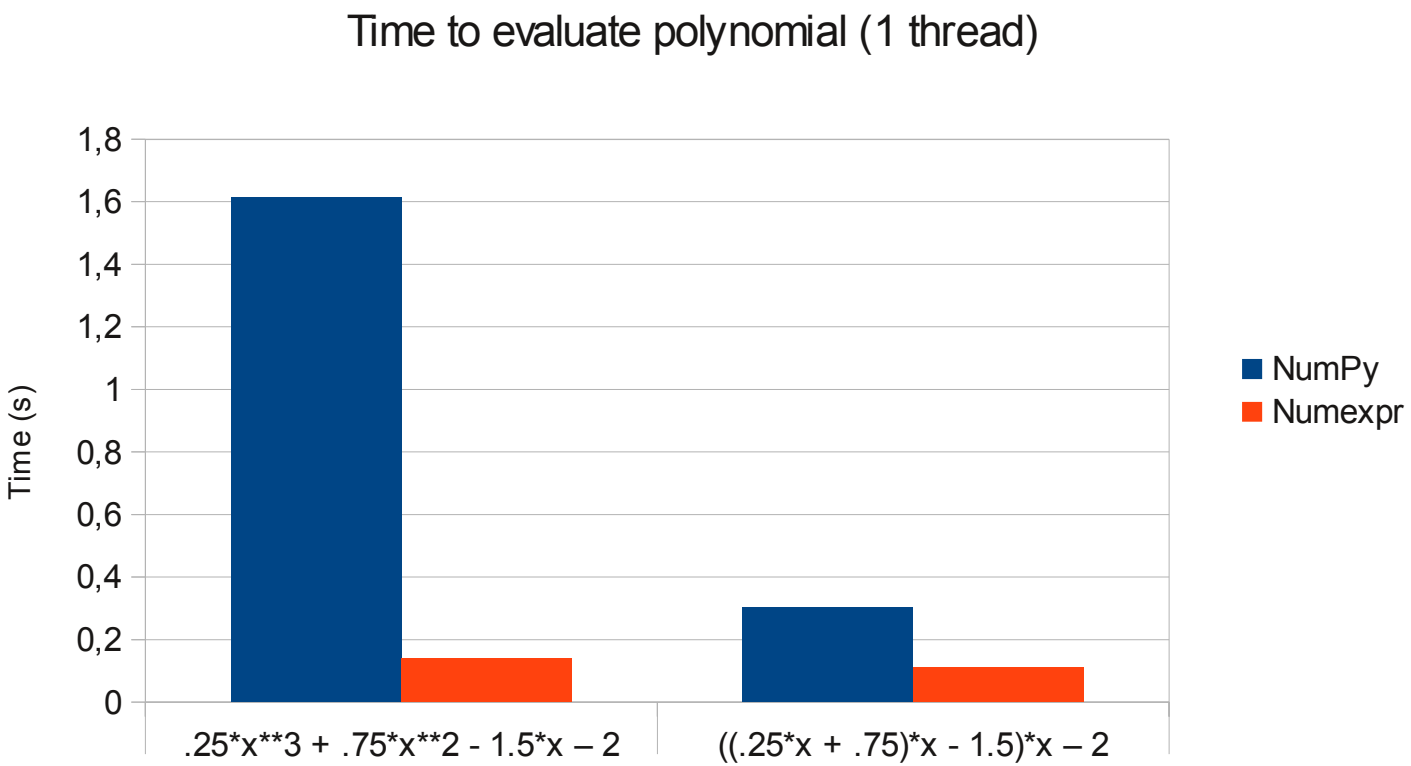
The Blocking Technique

When accessing disk or memory, get a contiguous block that fits in CPU cache, operate upon it and reuse it as much as possible.

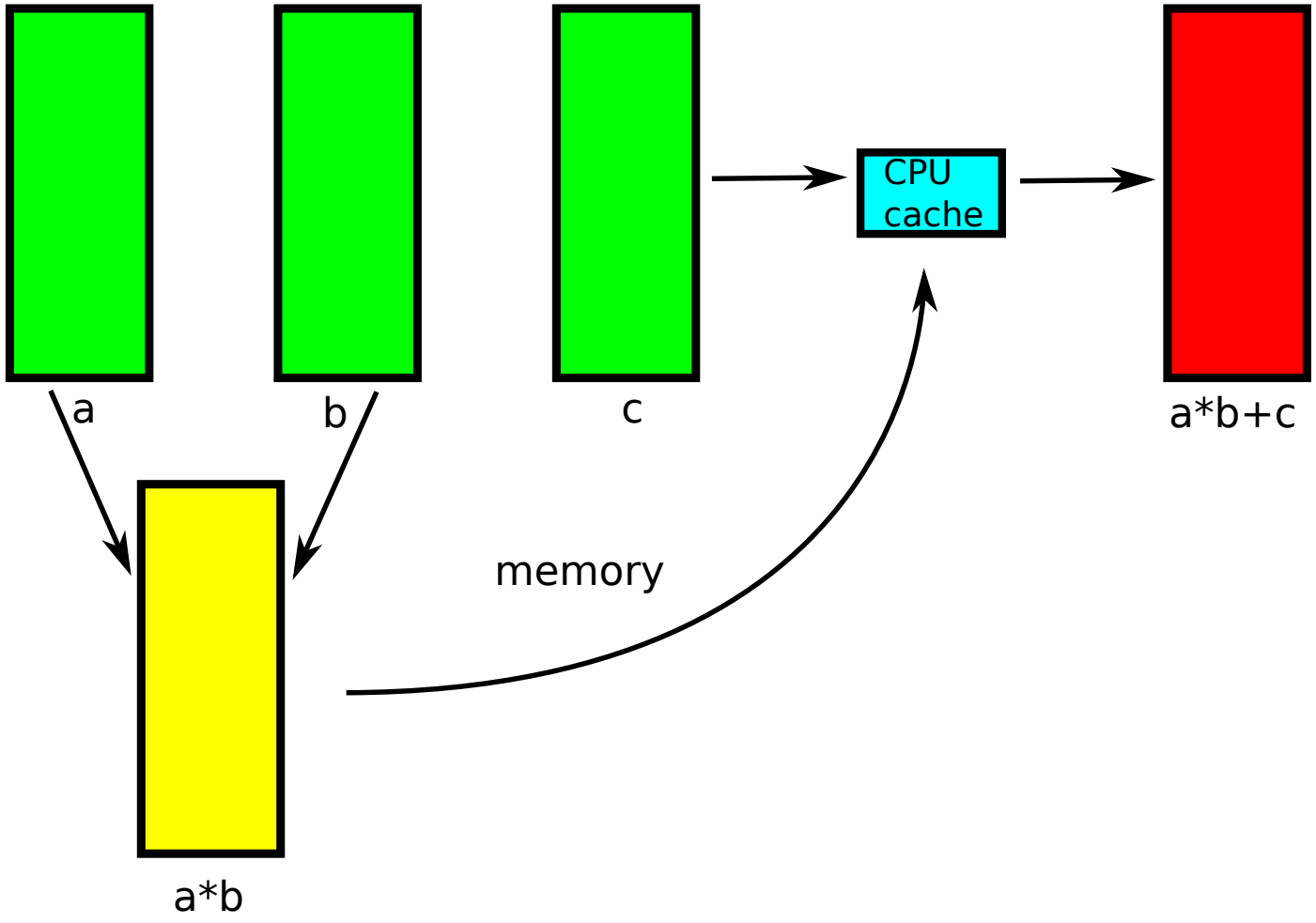


Use this extensively to leverage spatial and temporal localities

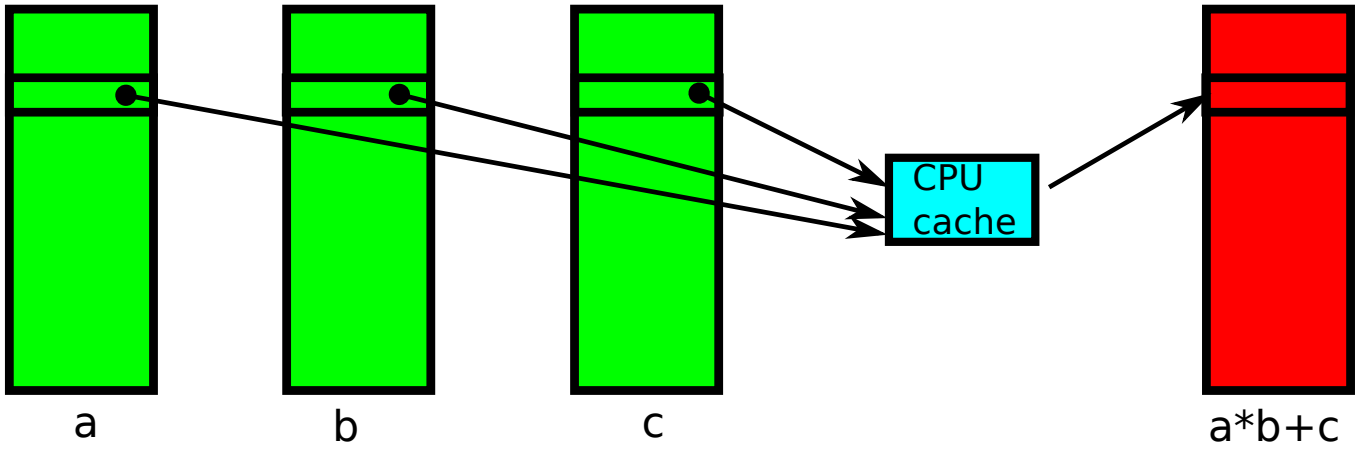
Time To Answer Pending Questions

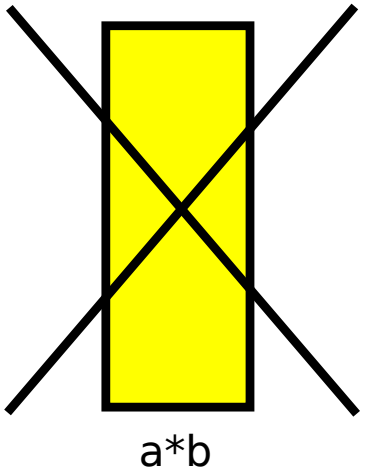


Computing "a*b+c" with NumPy. Temporaries goes to memory.



Computing "a*b+c" with Numexpr. Temporaries in memory are avoided.





memory

Second Message of the Day

- Before spending too much time optimizing by yourself make you a favor:

  Use the existing, powerful libraries out there

  It is pretty difficult to beat performance professionals!

Optimal Containers for Big Data

The Need for a Good Data Container

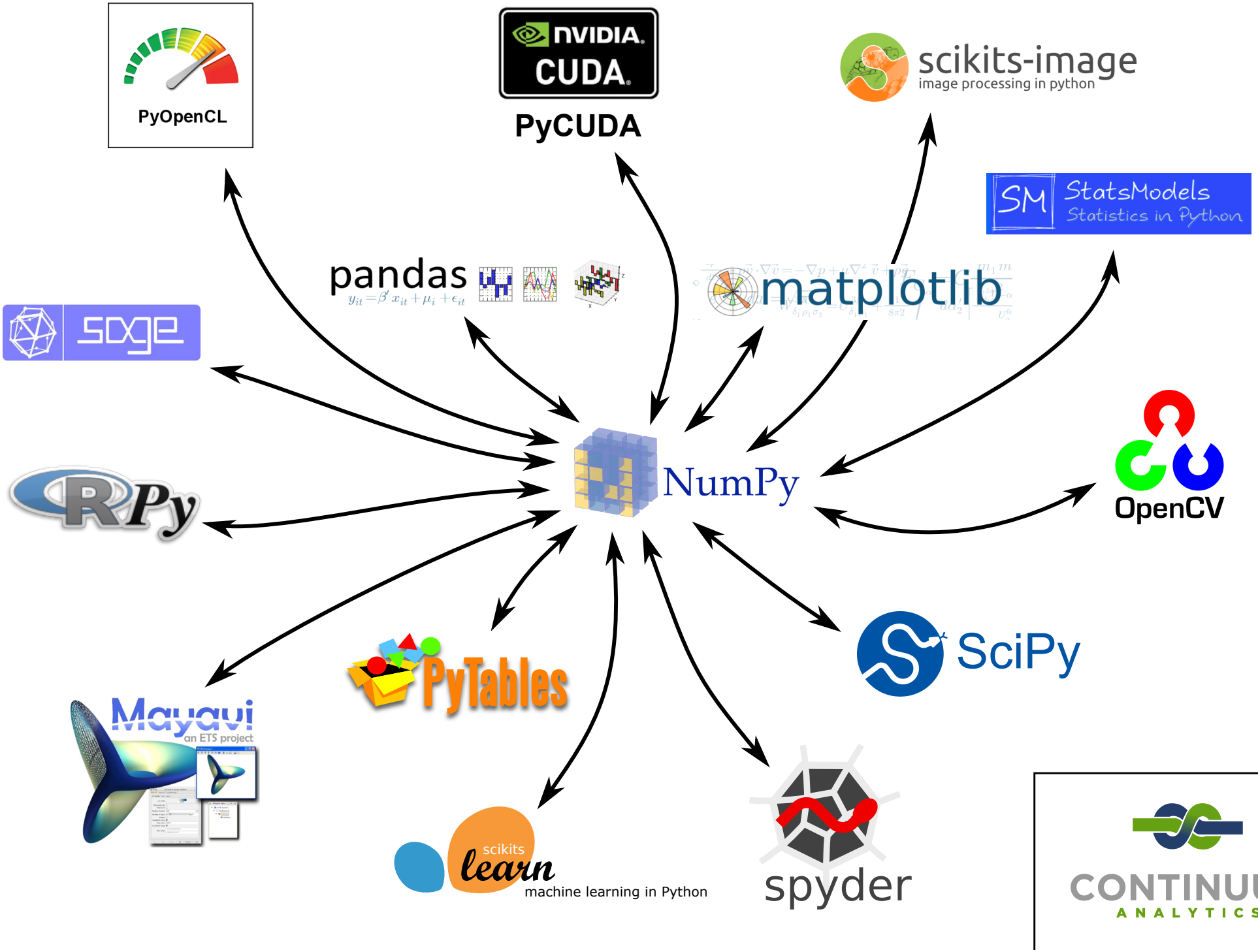
- Too many times we are focused on computing as fast as possible
- But we have seen how important data access is
- Hence, having an optimal data structure is critical for getting good performance when processing very large datasets

No Silver Bullet

- Unfortunately, there is not (and probably will never be) a fit-all data container
- We need to make our mind to the fact that we need to choose the 'optimal' container for **our case**

NumPy: A De Facto Data Container

NumPy is the standard de facto in-memory container for Big Data applications in the Python universe


PyOpenCL
NVIDIA
CUDA
PyCUDA
scikits-image
image processing in python
StatsModels
Statistics in Python
pandas
matplotlib
sage
NumPy
RPy
OpenCV
PyTables
SciPy
Mayavi
an ETS project
scikits
learn
machine learning in Python
spyder
CONTINUUM
ANALYTICS

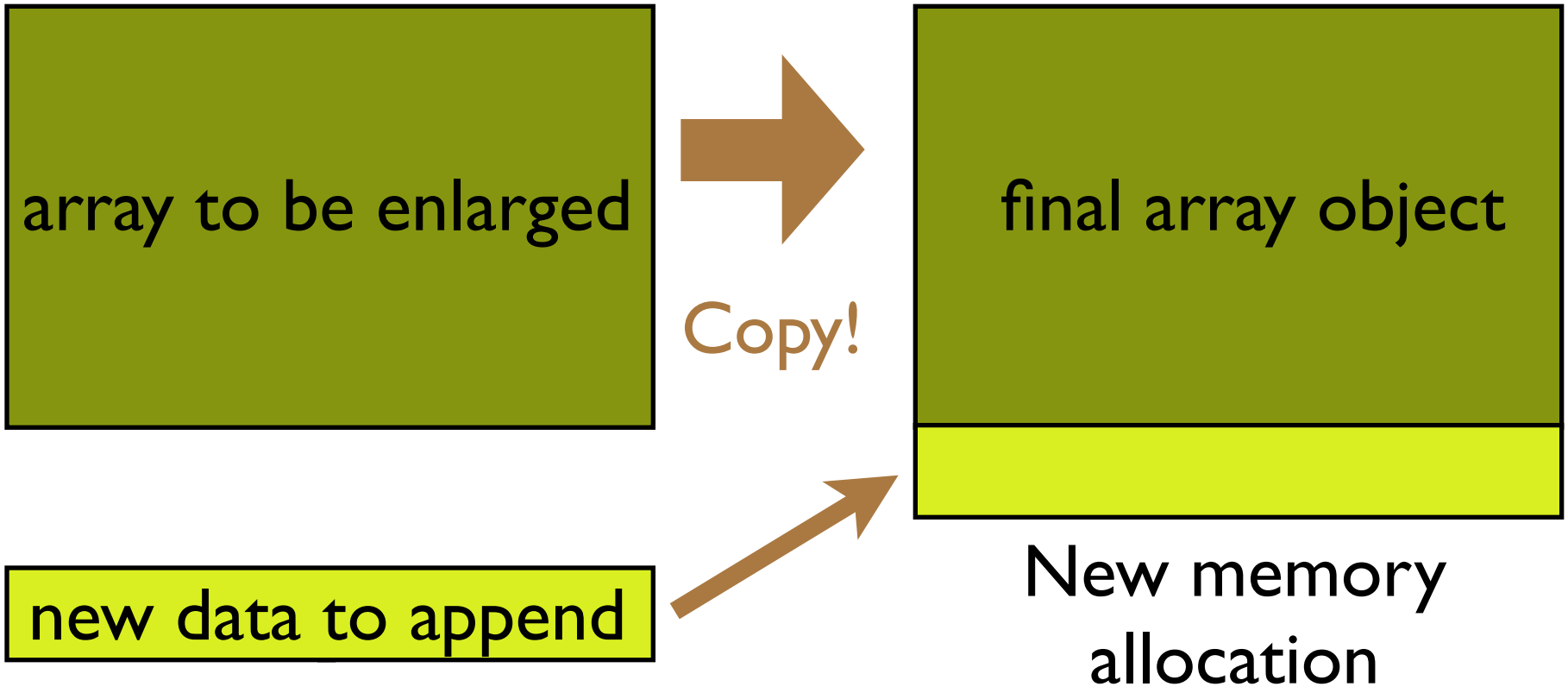
NumPy Advantages

- Multidimensional data container
- Efficient data access for many scenarios
- Powerful weaponry for data handling
- Efficient in-memory storage

Nothing Is Perfect

- The NumPy container is just great for many use cases
- However, it also has its own deficiencies:
  - Not efficient for appending data (so data containers tend to be **static**)
  - Cannot deal with compressed data transparently
  - Limited disk-based data support

Appending Data in Large NumPy Objects



- Normally a `realloc()` syscall will not succeed
- Both memory areas have to exist **simultaneously**

bcolz
Overcoming NumPy Limitations

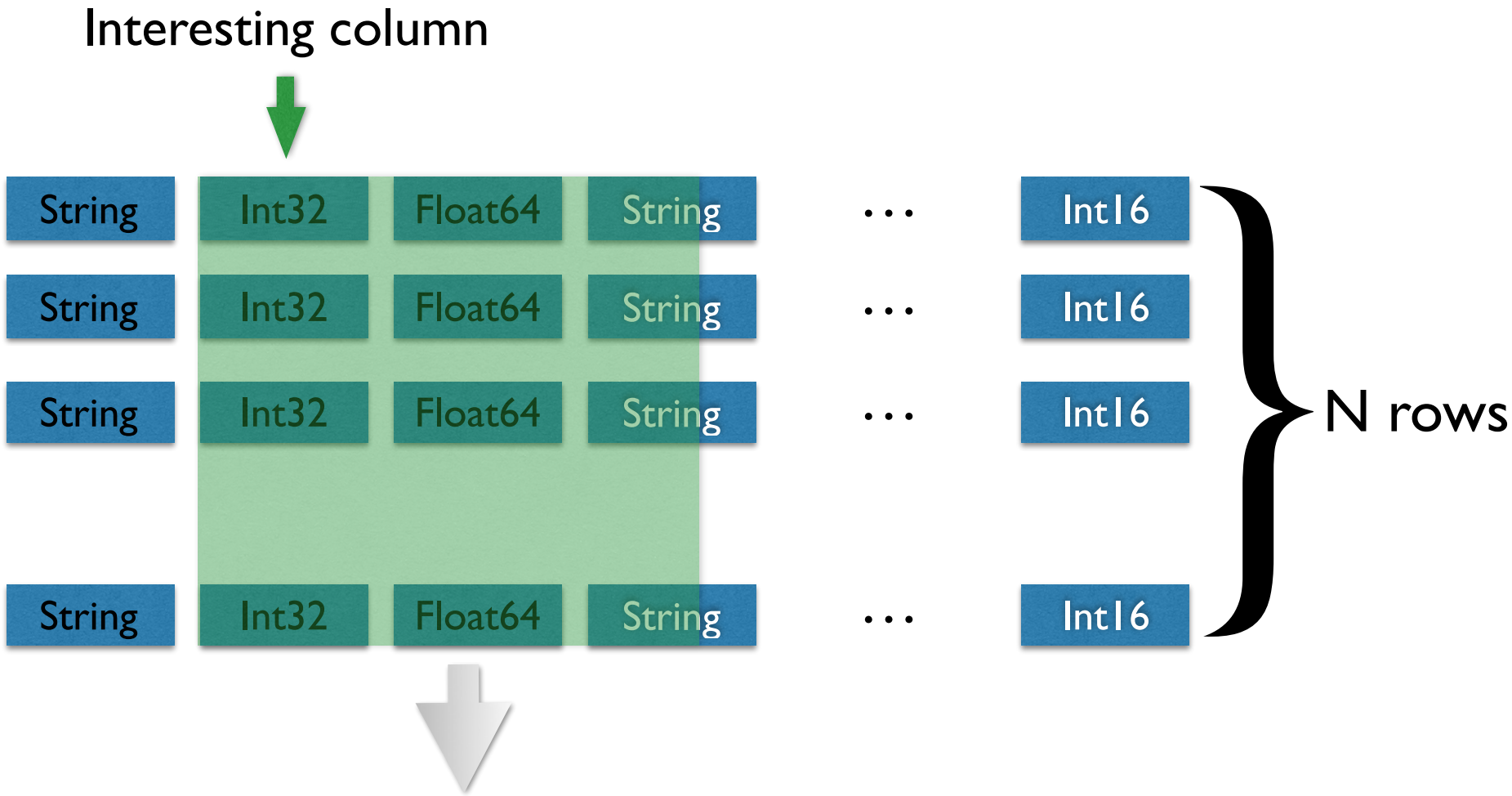
bcolz

- **Columnar, chunked, compressed** data containers for Python
- Offers **carray** and **ctable** container flavors
- Implements just a **few simple, but fast iterators** over the containers, supporting query semantics
- Uses the powerful **Blosc** compressor under the hood

Why Columnar?

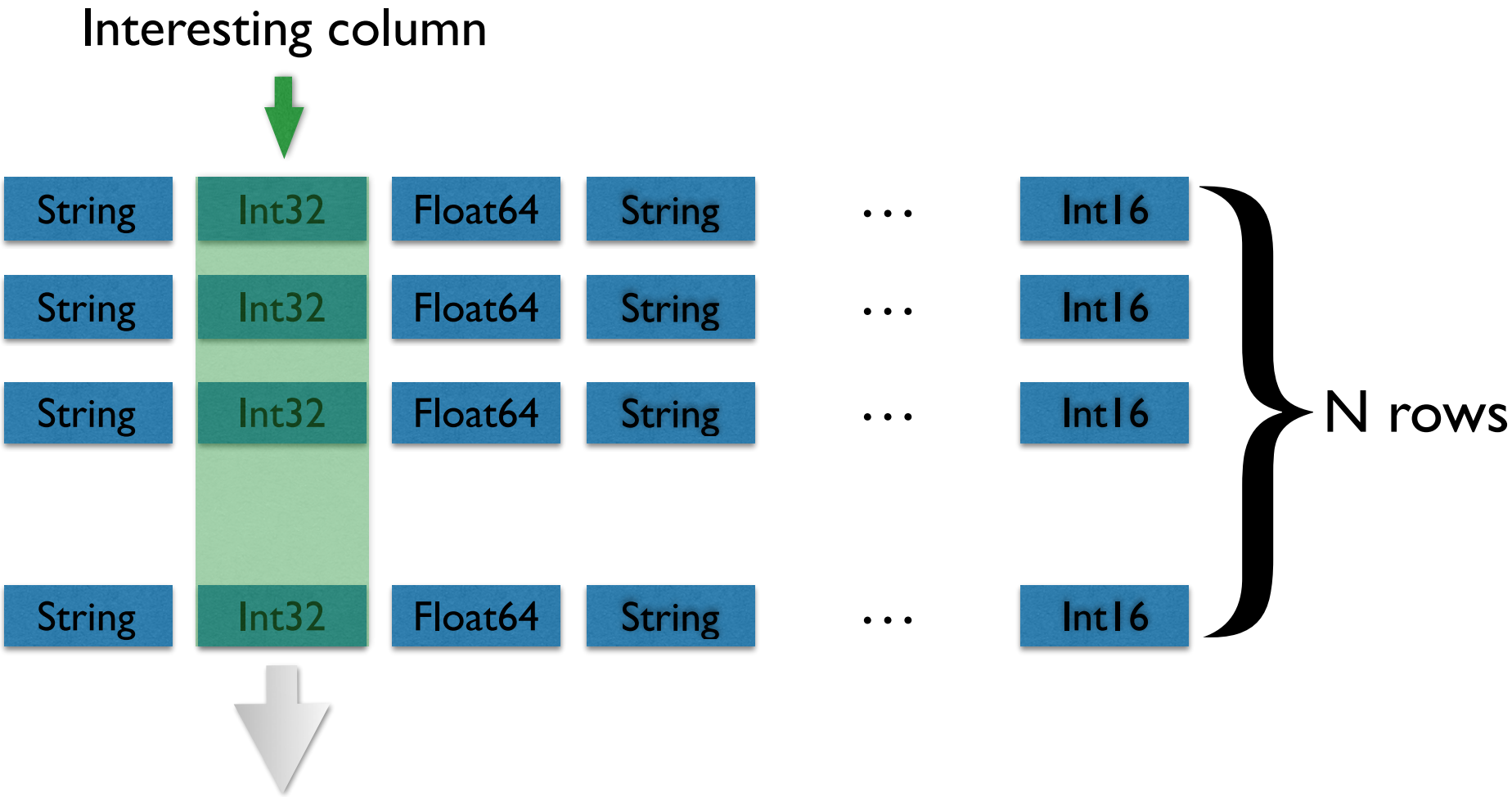
- When querying tabular data, only the **interesting** data is accessed
- Less I/O required

In-Memory Row-Wise Table



Interesting Data: N * 4 bytes (Int32)
Actual Data Read: N * 64 bytes (cache line)

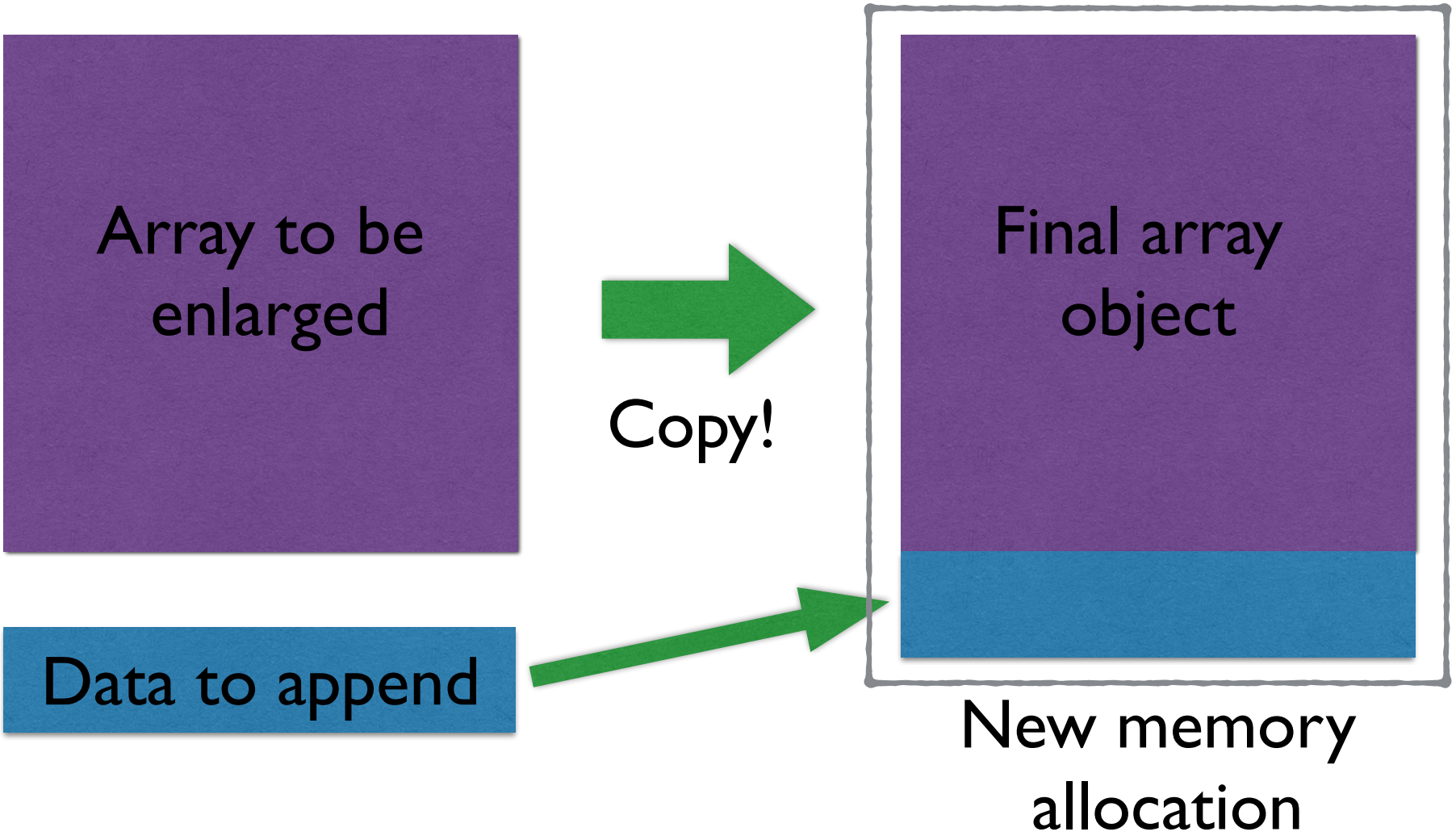
In-Memory Column-Wise Table



Why Chunking?

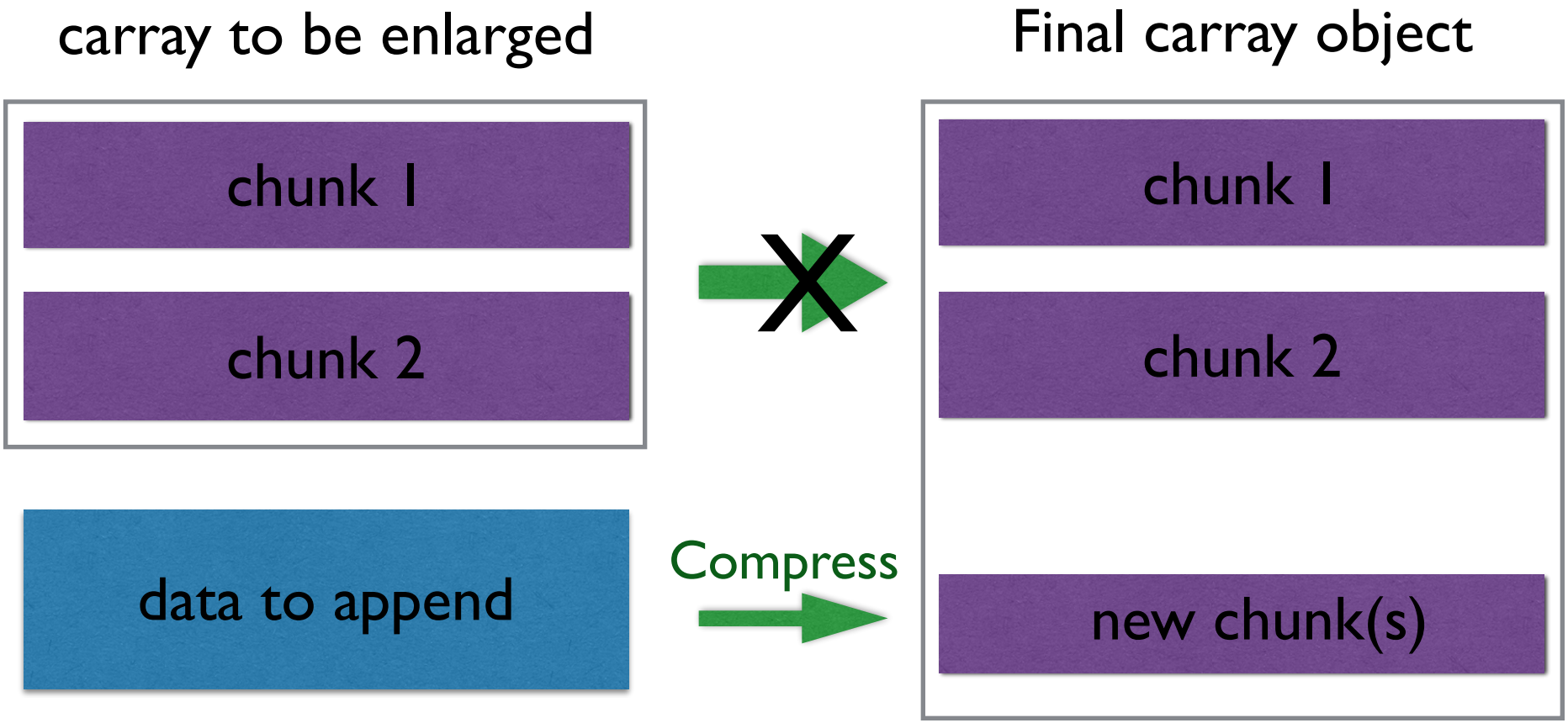
- Chunking means more difficulty handling data, so why bother?
  - Efficient enlarging and shrinking
  - On-flight compression possible

Appending Data in NumPy



- Both memory areas have to exist **simultaneously**

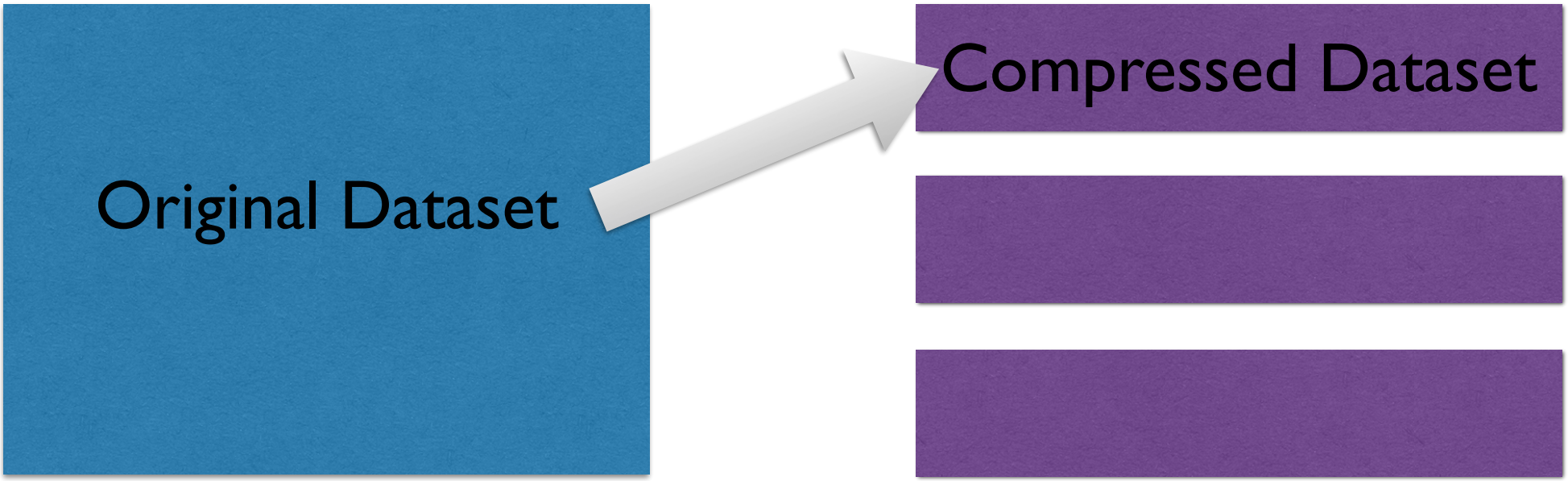
Appending Data in bcolz



- Only a compression operation on new data is required

Why Compression (I)?

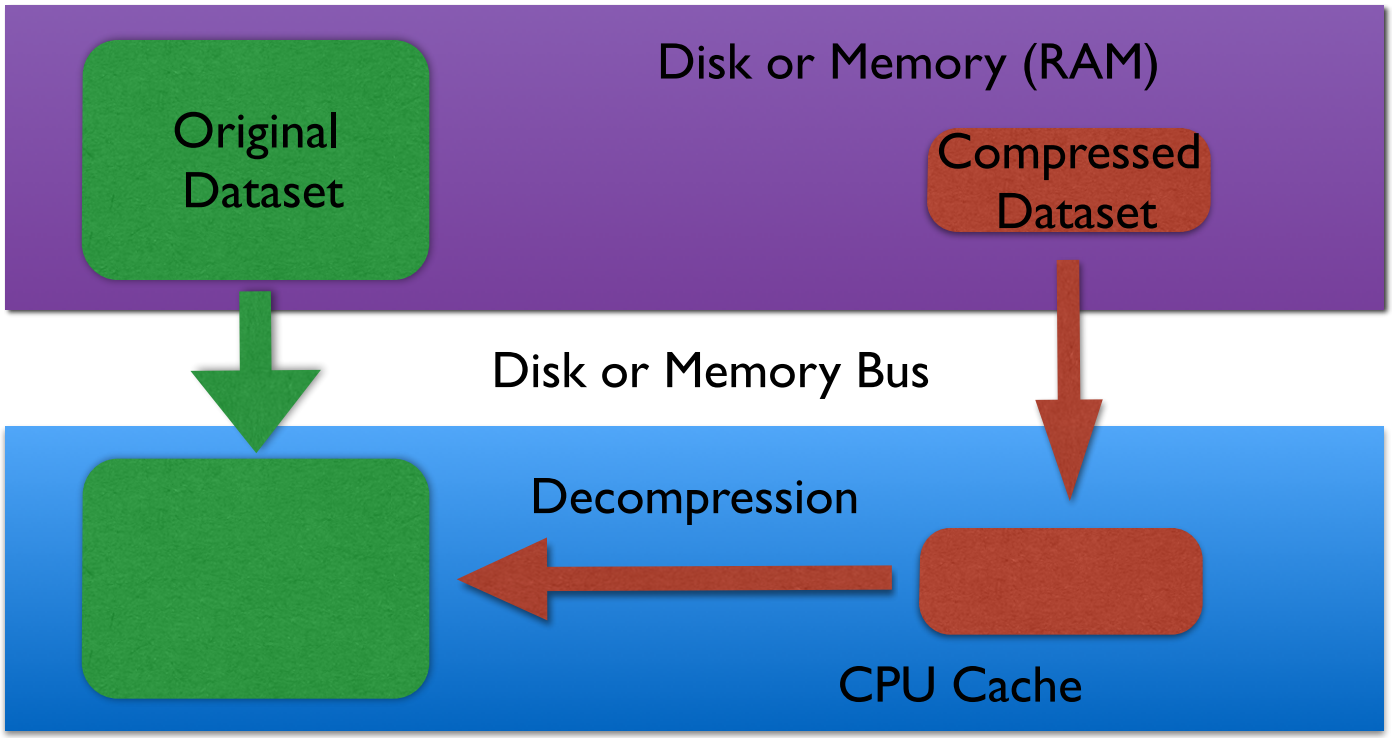
More data can be stored in the same amount of media



3x more data

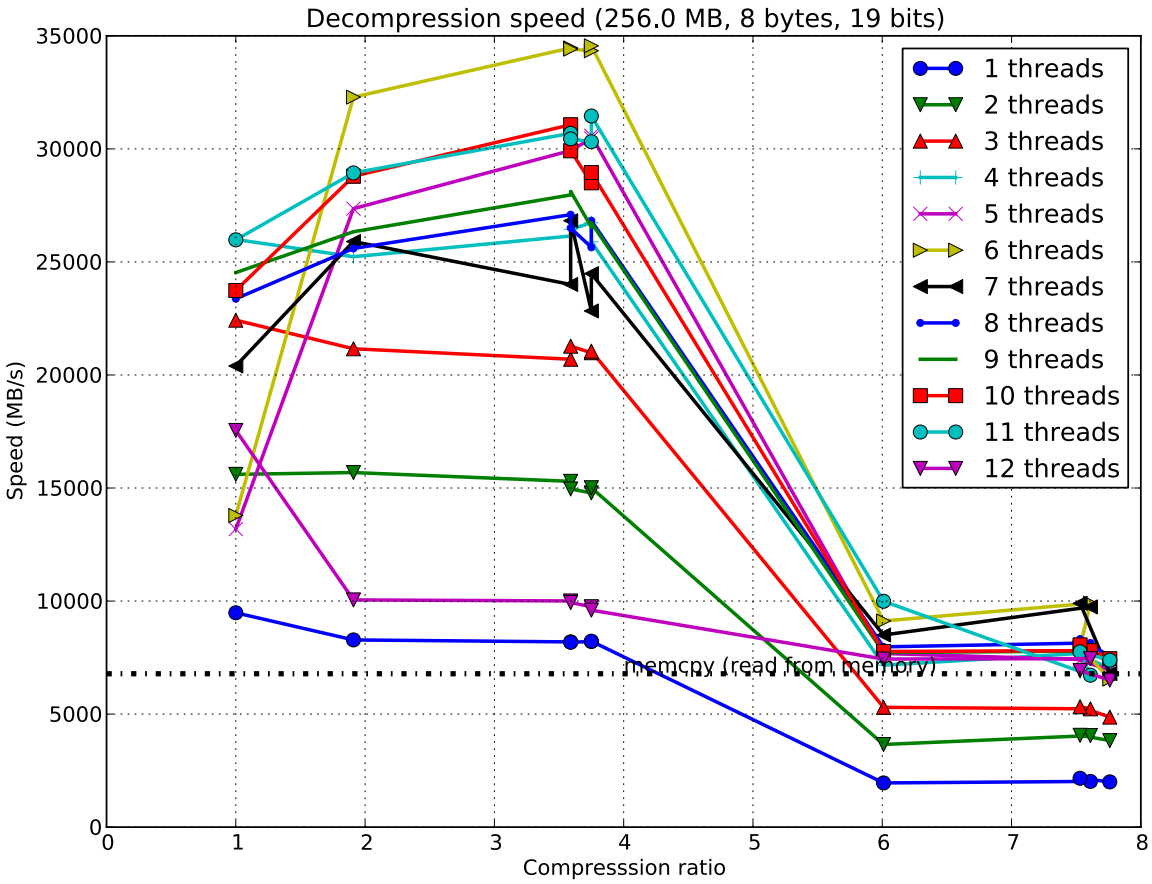
Why Compression (II)?

Less data needs to be transmitted to the CPU

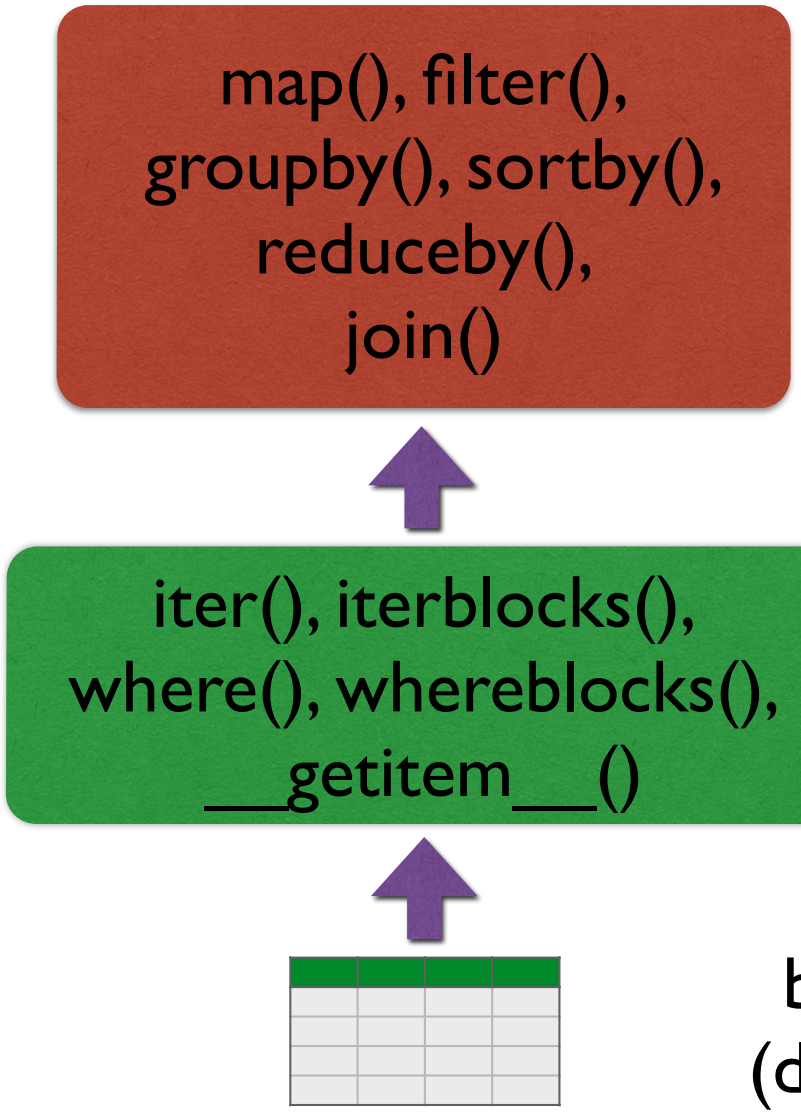


Transmission + decompression faster than direct transfer?

Blosc: Compressing Faster Than Memory Speed



Streaming Analytics with **bcolz**



itertools,
PyToolz,
CyToolz

bcolz
iterators/filters
with blocking

bcolz container
(disk or memory)

Some Benchmarks With Real Data: The MovieLens Dataset

Materials in:
https://github.com/Blosc/movielens-bench

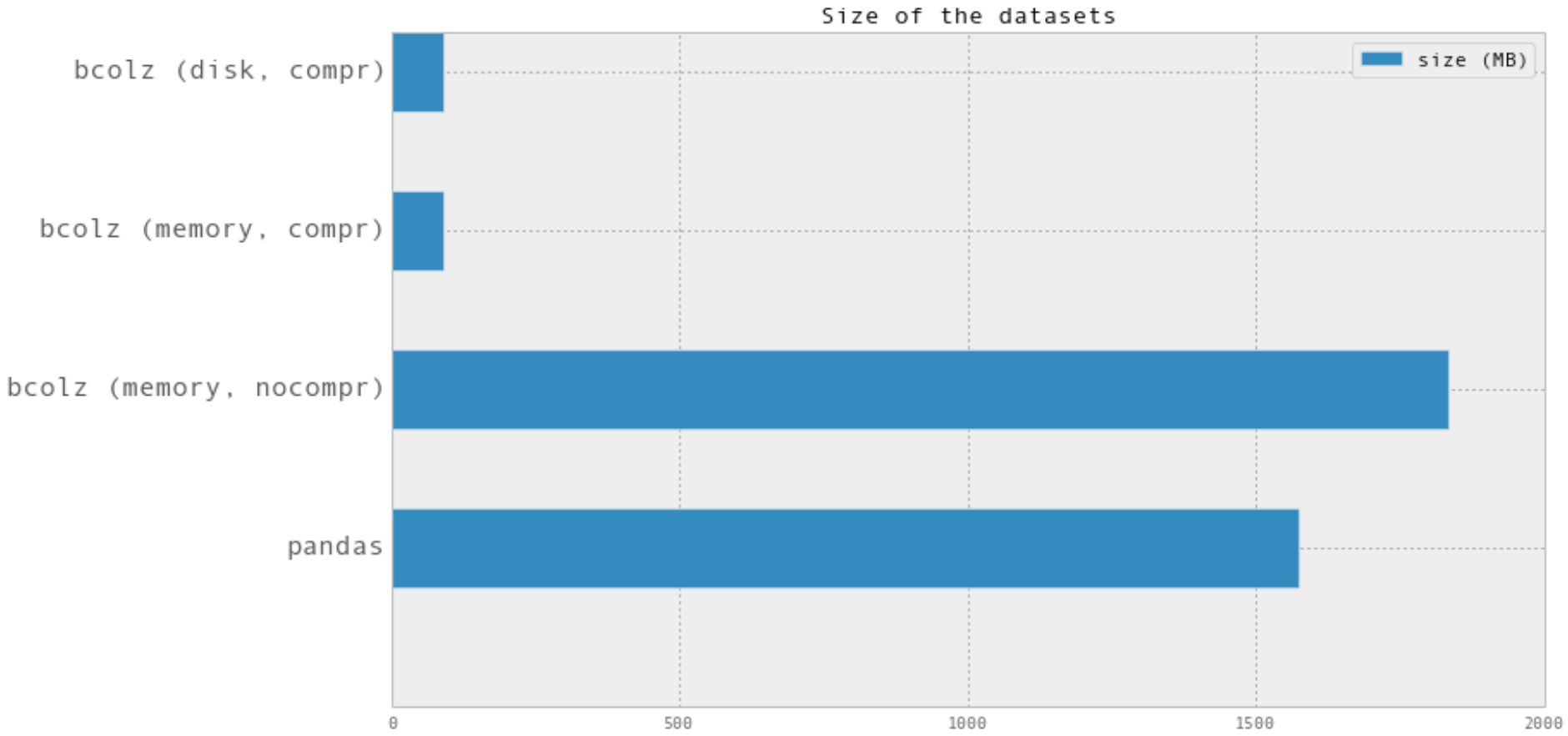
The MovieLens Dataset

- Datasets for movie ratings
- Different sizes: 100K, 1M, 10M ratings (the 10M will be used in benchmarks ahead)
- The datasets were collected over various periods of time

Querying the MovieLens Dataset

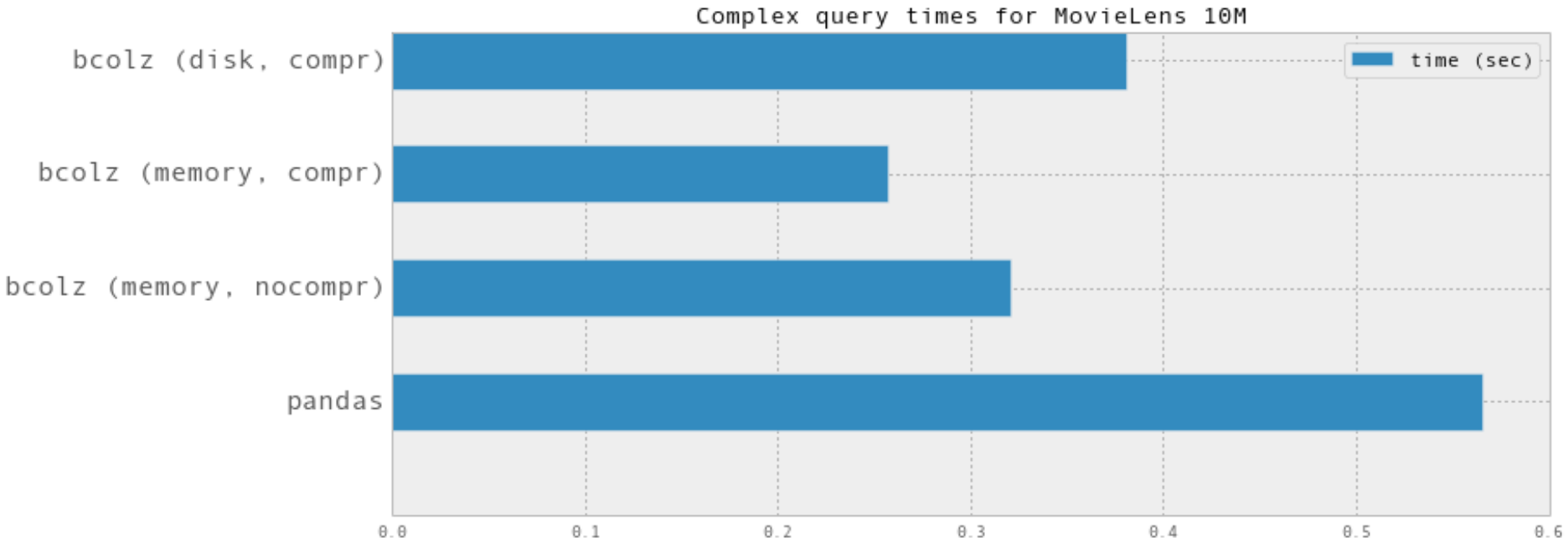
```
import pandas as pd
import bcolz

# Parse and load CSV files using pandas
…

# Merge some files in a single dataframe
lens = pd.merge(movies, ratings)

# The pandas way of querying
result = lens.query("(title == 'Tom and Huck (1995)') & (rating == 5)”)
[‘user_id']

# Get a ctable container
zlens = bcolz.ctable.fromdataframe(lens)

# The bcolz way of querying (notice the use of the `where` iterator)
result = [r.user_id for r in dblens.where(
    "(title == 'Tom and Huck (1995)') & (rating == 5)", outcols=['user_id'])]
```

Size of Datasets



- Compression means ~20x less space
- The uncompressed ctable is larger than pandas

bcolz Query Performance



- Compression leads to better query speeds (15% faster)
- Querying a disk-based ctable is fast!

Final Take Away Message for Today

- Big data is tricky to manage:

  Look for the optimal containers for your data

  Spending some time choosing your appropriate data container can be a big time saver in the long run

Summary

- Nowadays you should be aware of the memory hierarchy for getting good performance
- Leverage existing memory-efficient libraries for performing computations optimally
- Use the appropriate data containers for your different use cases

References

- Why Modern CPUs Are Starving And What You Can Do About It
- bcolz: http://github.com/Blosc/bcolz
- Blosc ecosystem: http://blosc.io

Thank you!

Questions?

francesc@blosc.io
@FrancescAlted