High-Performance Data Science with Python

—— (2) DataFrame Game ——

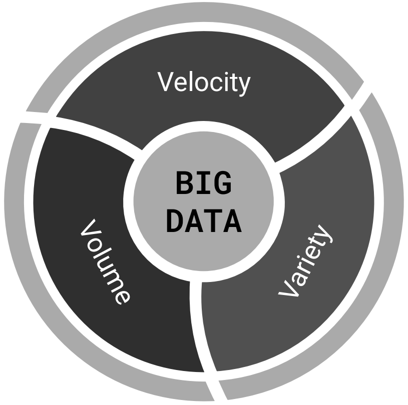
Qiyang Hu

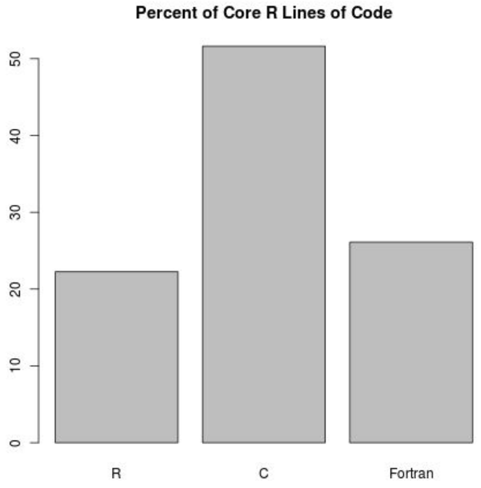
UCLA IDRE/OARC Workshop

May 12th, 2021

Recap from the last lecture

- We discussed “Velocity” in big data analytics.
    - Most optimizations were done by the compiler(JIT, AOT).
    - Today we will address “Volume” and “Variety”.
    - Dataframe may have implementation of JIT to speed up.
- We focus on high-level reviews
    - End-user oriented
- A side note about the R question:
    - R is the interpretive language.
    - Core of R is written by C, Fortran and R
    - R is strongly and dynamically typed.
    - R has its own JIT packages
        - jit, compiler, ...





Key issues in dealing with array-type data

- Data types
  - Fixed-width vs. variable-width: numericals, strings
  - Nullable/masked: can/cannot be None
  - Heterogeneous: different types in an array
  - Nested records with named (dict) or unnamed (tuple) fields
- Data structures
  - Structured: numpy, pandas, ...
  - Unstructured:
    - Tree structures: datrie, treelib, awkward arrays, ...
    - Graph structures: networkx, stellar graph, ...
- Data storage
  - Memory & Disk format: row-based vs columnar
  - Sparseness: dense vs sparse
  - Chunking and partitioning: for parallel processing
- Data manipulation
  - Virtualness: lazy load/evaluation
  - Subclassing arrays with high-level specialized methods

In most cases, numpy serves as a building block:

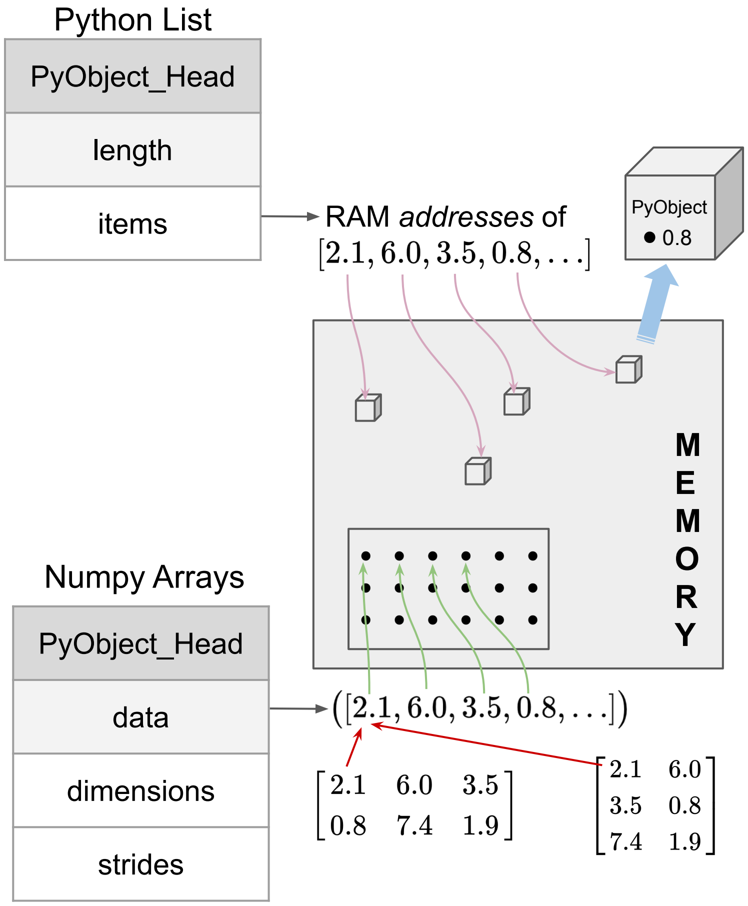
- Using numpy tricks to manipulate
- Use numpy's extension module to dispatch its own func
- Overrides numpy's ufuncs

Outline

- Vanilla arrays and dataframes
    - NumPy
    - Pandas
- Parallel processing with multi-cores
    - Pandallel
    - Modin
    - Swifter
- Out-of-core approaches
    - Dask Arrays and Dataframes
    - Vaex

NumPy

- First choice for numerical arrays
  - Overall 5 to 100 times faster than Python list.
- Internals of numpy vs. Python list
  - Python list: scattered across the system memory
  - Numpy data: stored in a continuous block memory
    - Array computation part re-written in C
    - Performance gain due to CPU caching
    - Vectorized instructions of modern CPUs
    - Can be linked to BLAS, MKL, etc.
- Performance tips
  - Use *view*, avoid *copy* (watch out for implicit-copy!)
  - Take advantage of vectorization
    - numpy.vectorize()
    - use broadcasting if possible



pandas DataFrame

bit.ly/hpdspy_02

- Under the hood
  - It groups the columns into blocks of values of the same type.
  - It represents numeric values as NumPy ndarrays and stores them in a continuous block of memory.
  - The object type represents values using Python string objects
- Optimize the numeric data
  - int64 and float64 are default and expensive
  - Subtypes can save RAM and a bit faster
- Optimize the object data
  - Strings are expensive and slow
  - Categoricals: 10x to save RAM & speedup

DataFrame

	date	number_of_game	day_of_week	v_name	v_league	v_game_number	h_name	h_league	length_outs
0	01871054	0	Thu	CL1	na	1	FW1	na	54.0
1	18710505	0	Fri	BS1	na	1	WS3	na	54.0
2	18710506	0	Sat	CL1	na	2	RC1	na	54.0

IntBlock

	0	1	2	3	4	5
0	01871054	0	1	1	0	2
1	18710505	0	1	1	20	18
2	18710506	0	2	1	12	4

ObjectBlock

	0	1	2	3	4
0	Thu	CL1	na	FW1	na
1	Fri	BS1	na	WS3	na
2	Sat	CL1	na	RC1	na

FloatBlock

	0
0	54.0
1	54.0
2	54.0

Try “dtype_diet” package to get help.

Optimize your pandas dataframe with its advice.

+

Drop to Numpy whenever you can.

Take advantage of vectorization in loops.

Pandaral·lel

- Numpy and Pandas: using single core
    - Workaround is possible, but very complicated
- Pandarallel: using multiple cores
    - Instantiates a Pyarrow Plasma shared memory
    - Creates one sub processes for each CPU to work on a sub part of the DataFrame
    - Combine all the results in the parent process
- A drop-in replacement:


```
from pandarallel import pandarallel
pandarallel.initialize()
df/series.apply -> df/series.parallel_apply
df/series.map -> df/series.parallel_map
df/series.applymap -> df/series.parallel_applymap
```

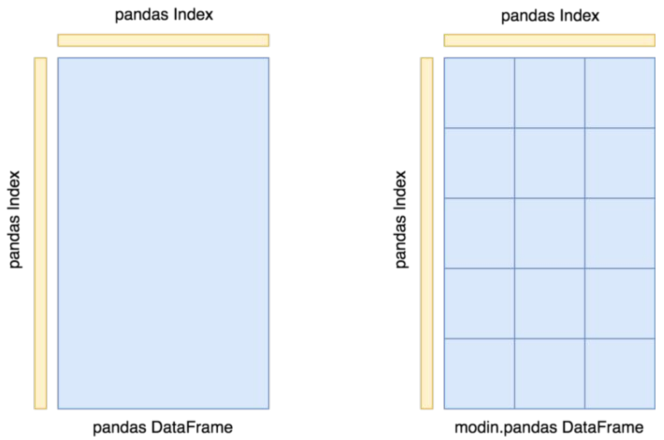
Watch out for the overhead!

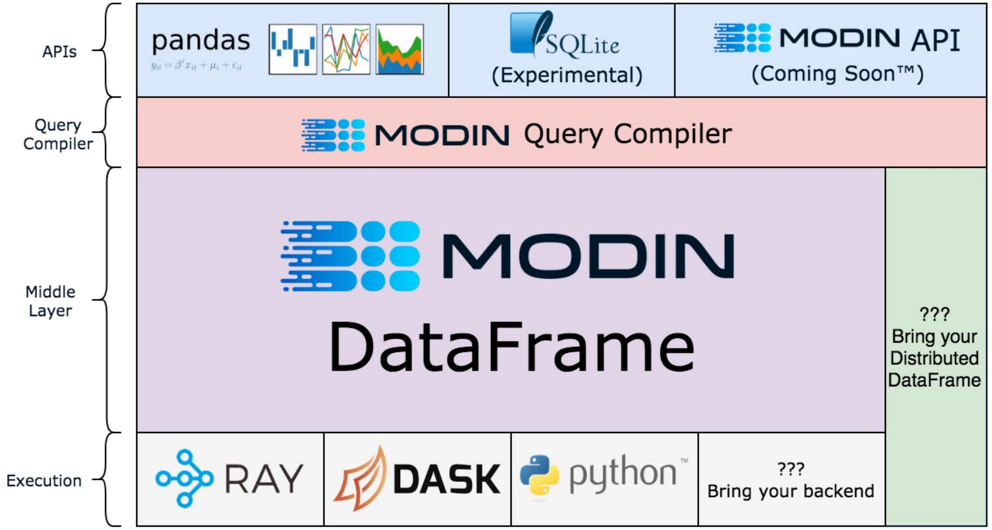
MODIN

- Multiprocess dataframe
  - Project from UCB's RISELab
  - Have identical APIs to Pandas
- Internals of Modin
  - 2-dimensional partitioning
  - Re-implementing Pandas APIs
  - 4-layer architectures
- Installation & Usage

```
pip install modin[dask/ray/all]
import os
os.environ["MODIN_ENGINE"]="dask"
```

```
import pandas as pd
```



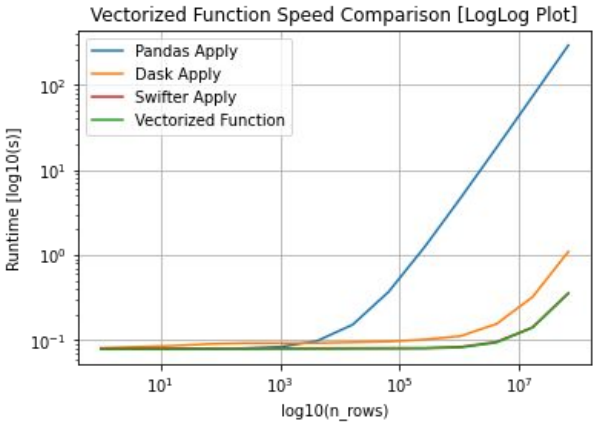


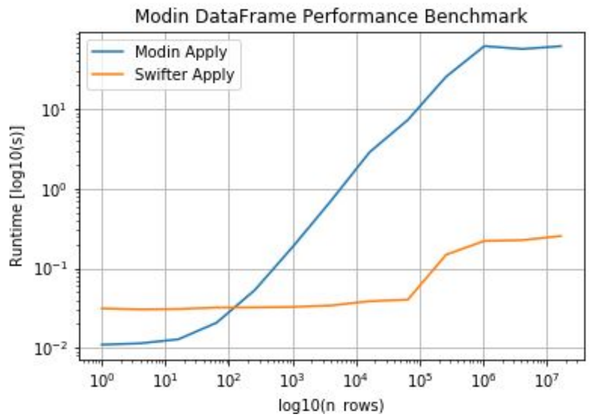
Swifter

- A booster to Pandas and/or Modins
  - A project developed by Jason Carpenter
- Basic idea:
  - Try vectorization first
  - If not succeeded, automatically decides whether it is faster to perform dask parallel processing or use a simple pandas apply.
- Installation and usage

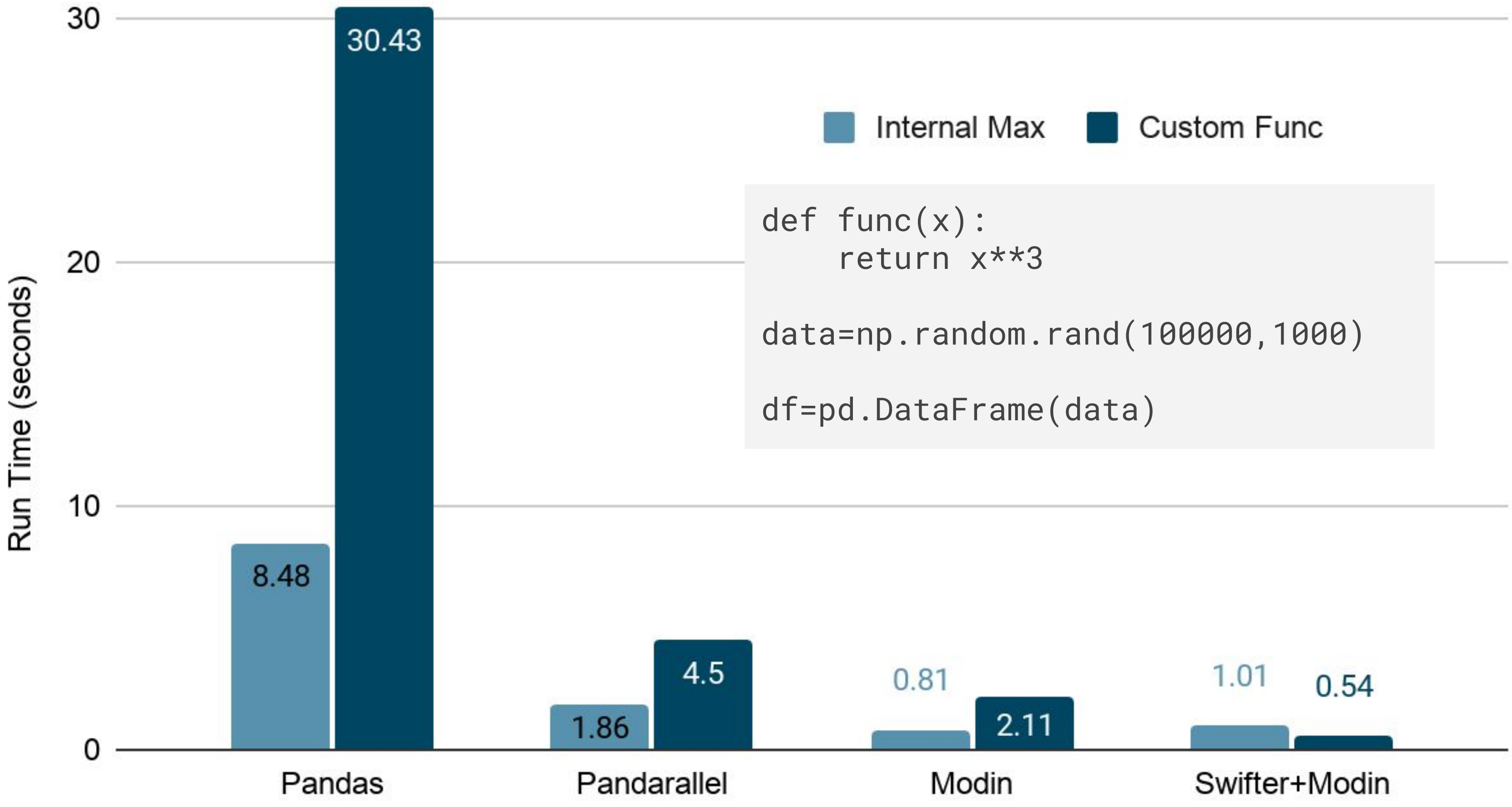
```
pip install swifter
import modin.pandas as pd
import swifter

df=pd.DataFrame(data)
df.swifter.apply(func, axis=1)
```



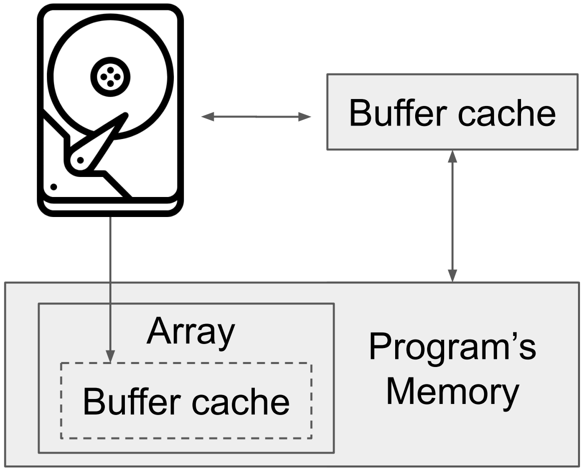


Dataframe's apply method speed test on Hoffman2



When Data > RAM

- Numpy Arrays
  - Using smaller subtypes
  - Compressing sparse arrays
  - Chunking the data for *on-demand* reads
    - Using `np.mmap()`
    - Using Zarr or HDF5 format

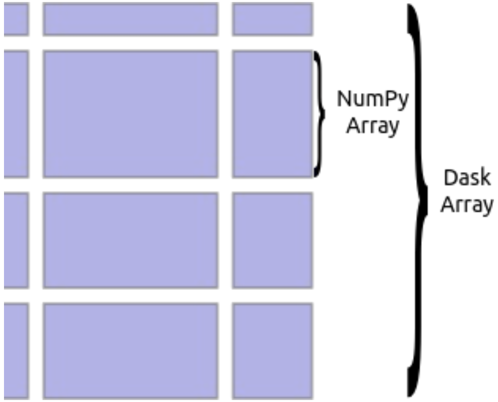


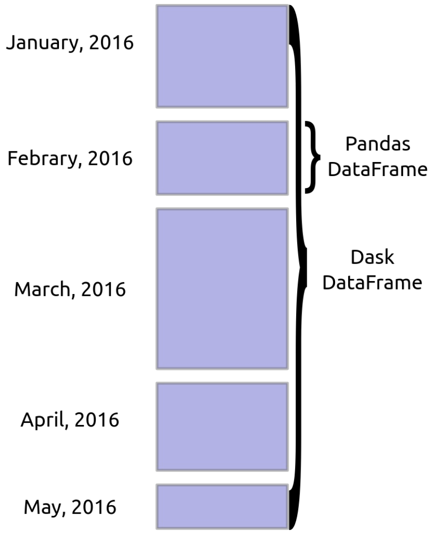
- Pandas Dataframes
  - Using Categorical dtypes and smaller subtypes
  - Loading columns selectively: `pd.read_csv(file, usecols=[...])`
  - Sampling rows: `pd.read_csv(file, skiprows=samplingFunc)`
  - Reading in chunks: `for df in pd.read_csv(file, chunksize=1000):`
  - Using SQLite as data storage for Pandas: 50x faster lookups

DASK Data Collections

- A full a parallel processing library suite
  - Working in parallel with ndarray and pd.DataFrame objects
  - Distributed computing with task schedulers to execute on clusters
  - Real-time feedback and diagnostics
- Three parallel data collections:
  - Dask Arrays: arranging ndarrays into chunks within a grid.
  - Dask DataFrames: partitioning pd.DataFrame along an index.
  - Dask Bags: for unstructured or semi-structured data
- Dasks APIs to emulate Pandas:
  - Setting up a static graph of operations first
  - Running the computation on that graph

```
import dask.dataframe as dd
df = dd.read_csv(file)
df.groupby(df.user_id).value.mean().compute()
```





vaex

bit.ly/hpdspy_04

- A promising open-source DataFrame library in Python
  - APIs closely resemble that of Pandas.
  - Easy to work with very large (>100G+) datasets efficiently
- Interesting features
  - Memory mapping
    - New string type (RAM efficient)
    - Support file formats of Apache Arrow, Apache Parquet, HDF5
    - Access from local disks and cloud storage
  - Dataframes are out-of-core
    - Lazy evaluations: compute only when needed (e.g. preview)
    - Lazy loading: data streaming only when needed to avoid memory copy
  - Parallelized and fast/efficient algorithms:
    - groupby, join, selection
    - JIT compilation: Numba, Pythran, CUDA
  - vaex.ml Package for *out-of-core* scikit-learn

When to use which?

		Multiple CPUs	Out-Of-Core	Pandas APIs	Keep-In-Mind Notes
1	Pandas	✗	✗	✓	• Always be the one if we can. • Lots of performance tricks!
2	Modin	✓	✓✗	✓	• Good for dataframes with many columns • Experimental out-of-core dataframes • Experimental distributed XG-Boost
3	Dask	✓	✓	✓	• Ultimate solution if distributed situations: ○ Data storage & computation. • Steep learning curve after a quick start
4	Vaex	✓	✓	✓✗	• Good for dataframes with many rows • HDF5 files work best so far. • Some APIs are different from Pandas

What we didn't talk about yet:

Spark DataFrame

Ray DataFrame

3rd session in "High Performance Data Science in Python"

RAPIDS cuDF

"Learning Scikit-Learn"

PyTorch Tensor

3rd session in "Learning Deep Learning with PyTorch"

Key Takeaways

- Use Pandas if it is possible
  - Drop to numpy if you can.
  - Try “dtype_diet” to save RAM for you.
- When to consider Modin:
  - Your machine have many CPUs and a lot of RAM
  - Your work depends on most Pandas operations.
  - Your data needs “groupby” on many columns.
- When to consider Vaex:
  - Your data has about 10M to 1B rows, <100 columns (according to Ian Ozsvald)
  - Your data is in HDF5 format on a single place (local or cloud).
  - You only need a small set of Pandas functions.
- When to consider Dask:
  - Your dataset needs to be distributed stored and operated over a cluster.