

Python for High Performance Data Analytics

—— (2) Data Arrays ——

Qiyang Hu

UCLA Office of Advanced Research Computing

May 5, 2023

Questions from last talk

- Q: Do you recommend a class or course for a beginner?
 - Python for Beginners - Learn Python in 1 Hour ([Youtube](#))
 - Software Carpentries workshops in the UC System ([Programming in Python](#))
- Q: Share a link or site of basic python information for visual designers?
 - GUI: PyQt, Kivy Designer, PyGTK
 - Visualization: Software Carpentries Workshops at UCLA
 - Animations: PyGame, Blender, ...
- Q: Would love a basic link / reference for python for statistics
 - Python build-in module: [statistics](#)
 - Third-party libraries: Numpy, Scipy, Statsmodels, Scikit-learn
- Q: Would it be possible to get this slide deck
 - Github Repo: <https://github.com/huqy/HighPerfDataSciPython>



COMPUTATION

Single Node/GPU, SIMD

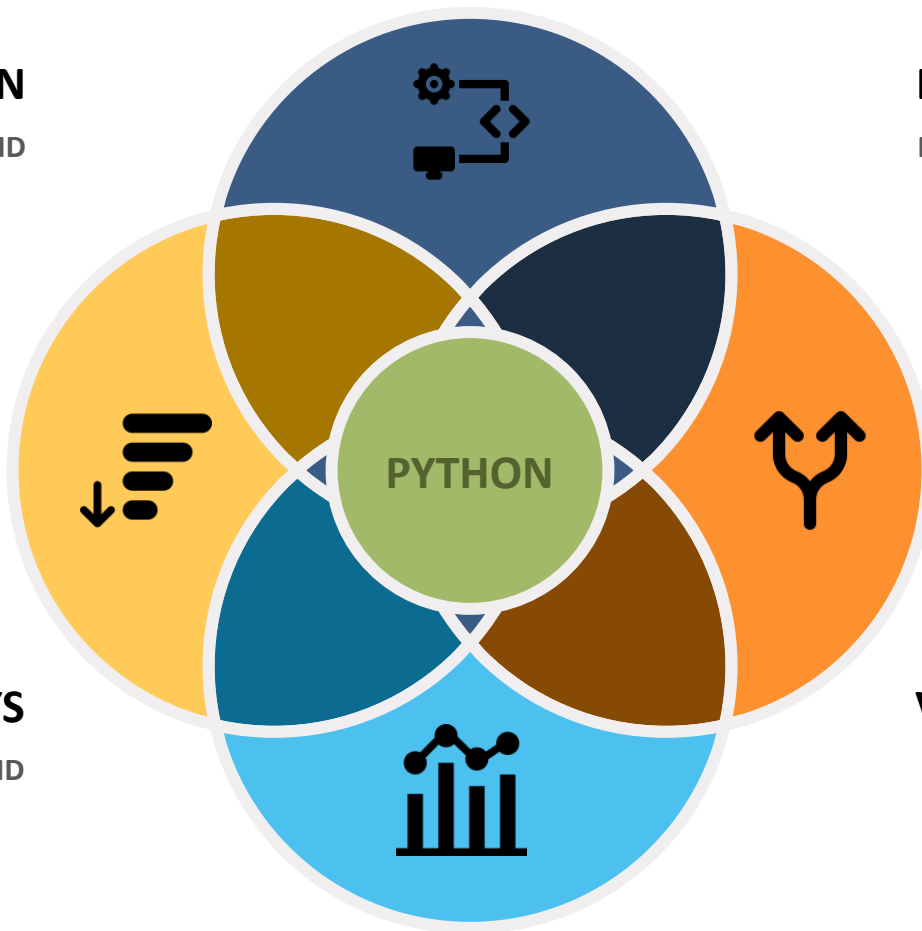
- Pypy, Numba, NumExpr
- Pythran, Cython
- F2py, ctypes



DATA ARRAYS

Single Node/GPU, SIMD

- Numpy
- Pandas, Polars
- Modin, Pandarallel, Swifter
- Dask DataFrame, Vaex



DISTRIBUTED



Multiple Nodes/Machines

- MapReduce-based: **PySpark**, PyFlink
- MPI-based: mpi4py, Horovod
- Joblib, Dask, Ray

VISUALIZATION



- Viz process for big data
- Matplotlib, Bokeh, Plotly
- Holoview and Datashader
- Traited VTK, Mayavi, Paraview

Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary

Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary

Key Issues in Array-Type Data

Data Types

- Fixed-vs.variable-width: numericals, strings
- Nullable/masked: can/cannot be None
- Heterogeneous: different types in array
- Nested records with named (dict) or unnamed (tuple) fields

Data Storage

- Memory & Disk format
- Row-based vs Columnar format
- Sparseness: dense vs sparse
- Chunking and partitioning: for parallel processing



Data Structures

- Structured: **numpy**, pandas, ...
- Unstructured:
 - nD-Panel data structure: xarray, [TensorStore](#)
 - Tree structures: datarie, treelib, awkward arrays, ...
 - Graph structures: networkx, stellar graph, ...

Data Processing

- Virtualness: lazy load/evaluation
- Subclassing arrays with high level specialized methods

Backends

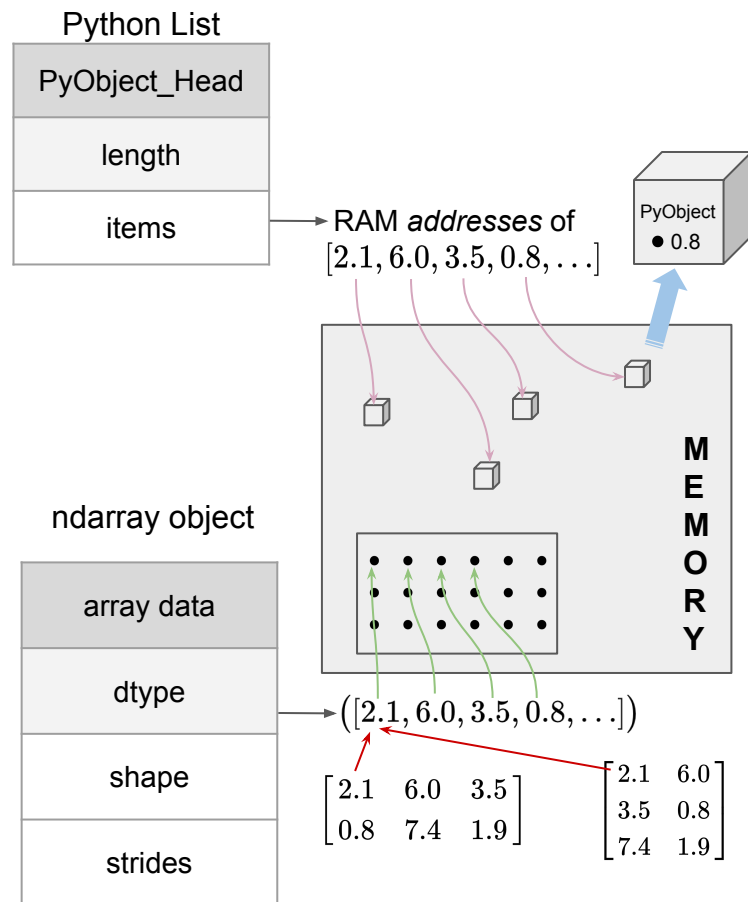
- Engines for storage, processing
- Two choices:
 - Numpy
 - Arrow

Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary



- First choice for numerical arrays
 - ndarray
 - Functions/tools for fast math and data ops
 - C APIs for C/C++, Fortran libs
- Internals of numpy vs. Python list
 - Python list: scattered across the system mem
 - Numpy data: stored in a continuous block mem
- Performance tips
 - Use *view*, avoid *copy*
 - watch out for implicit-copy!
 - Take advantage of vectorization
 - Try binary ufunc methods
 - Use broadcasting if possible
 - Explicitly define a vectorized function



pandas DataFrame v1.x

bit.ly/hpdsby_02

- NumPy and Pandas
 - Pandas built on top of numpy
 - Numpy: for homogeneously-typed numerical array data
 - Pandas: for tabular or heterogeneous data
 - Pandas provides more domain-specific functions
- Under the hood
 - Column grouping
 - Numeric values as NumPy ndarrays
 - Object type values as 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.

Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary

Pandarallel

- 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
```

```
In [40]: res = df.progress_apply(func, axis=1)
```

 100% 500000/500000 [00:22<00:00, 21916.87it/s]

```
In [38]: res_parallel = df.parallel_apply(func, axis=1)
```

 100% 125000/125000 [00:07<00:00, 16723.32it/s] 100% 125000/125000 [00:07<00:00, 16643.05it/s] 100% 125000/125000 [00:07<00:00, 16608.92it/s] 100% 125000/125000 [00:07<00:00, 17300.09it/s]

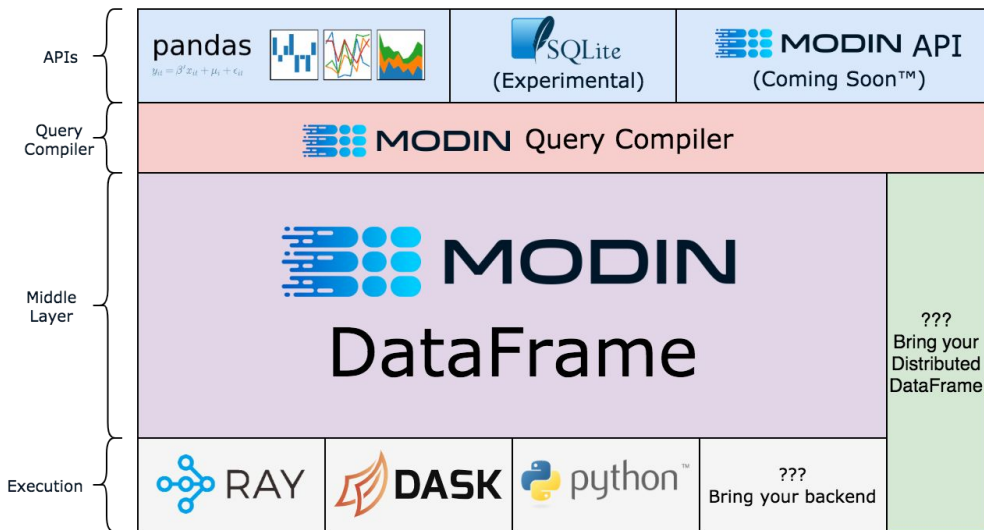
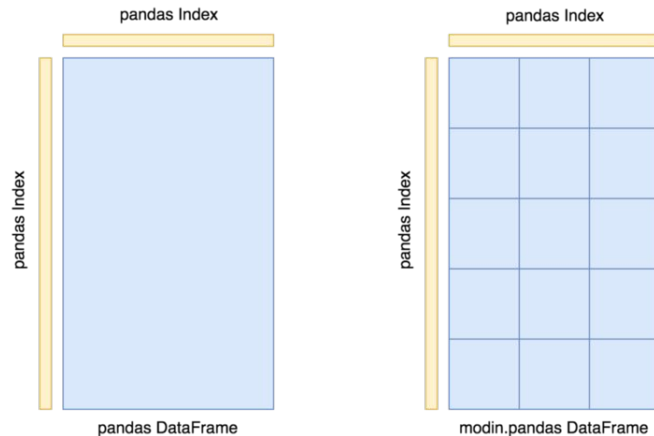
Watch out for the
overhead!



- 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
```

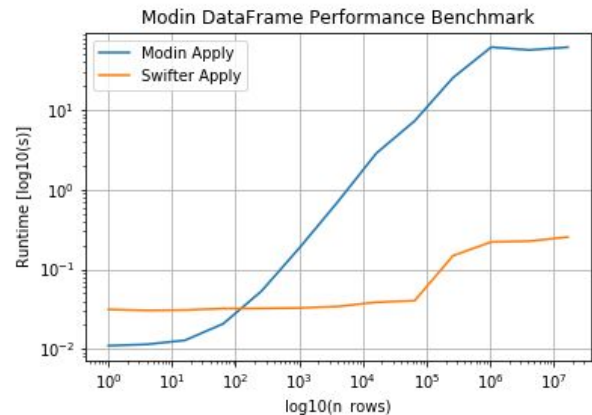
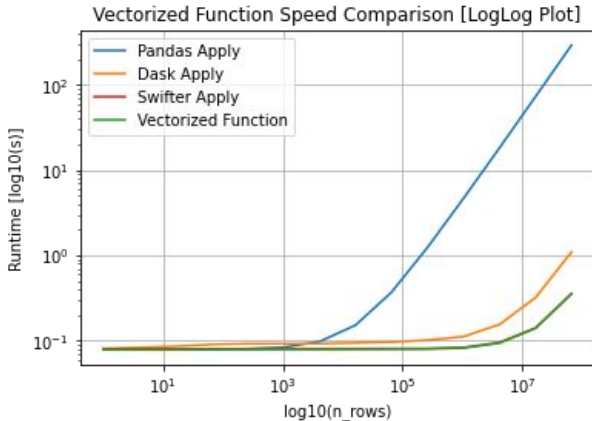


Swifter

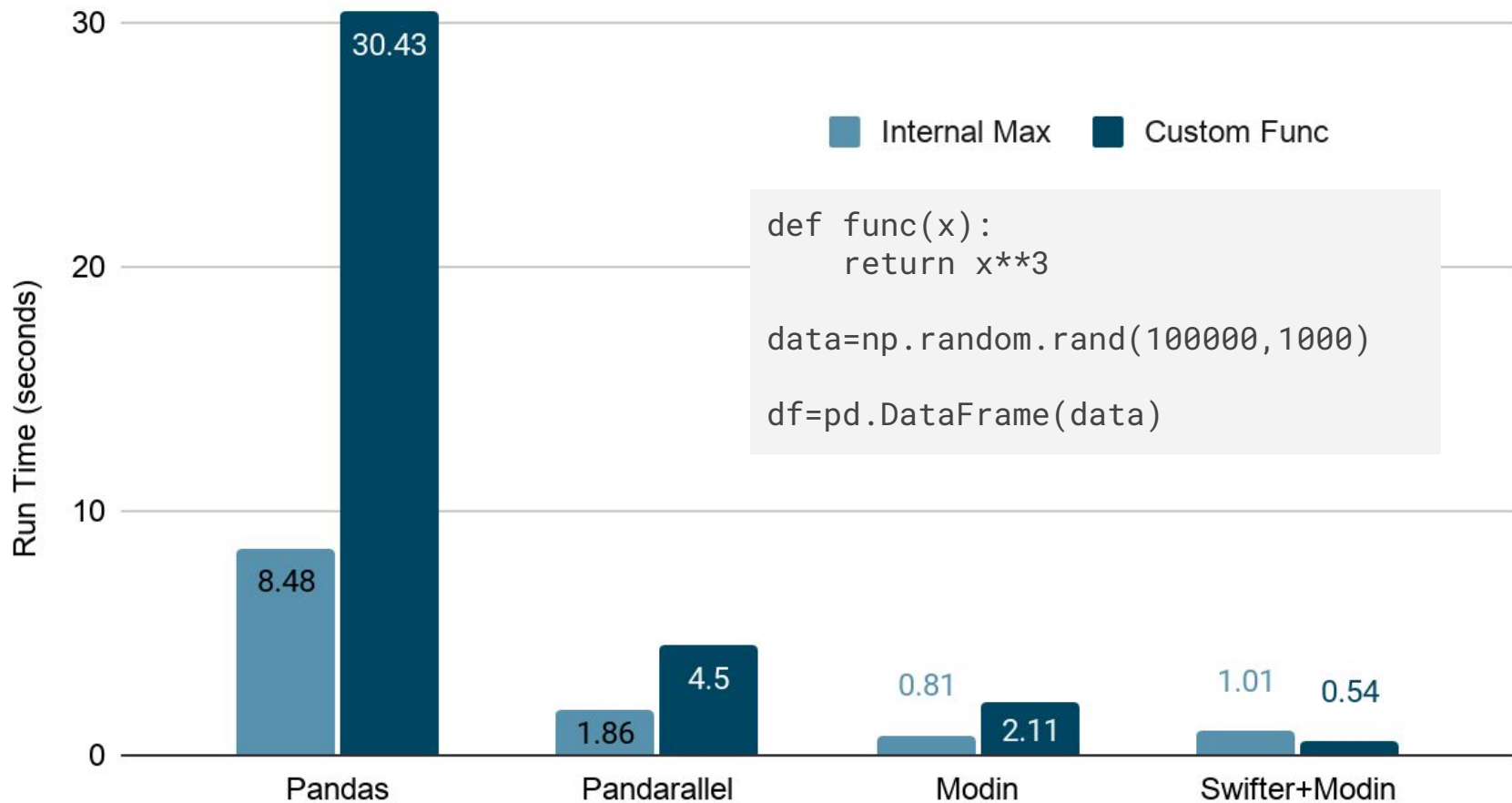
- A booster to Pandas and/or Modins
 - A project developed by Jason Carpenter
- Basic idea:
 - Try vectorization first
 - If not succeeded, automatically decides whether it is faster to perform dask parallel processing or use a simple pandas apply.
- Installation and usage

```
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



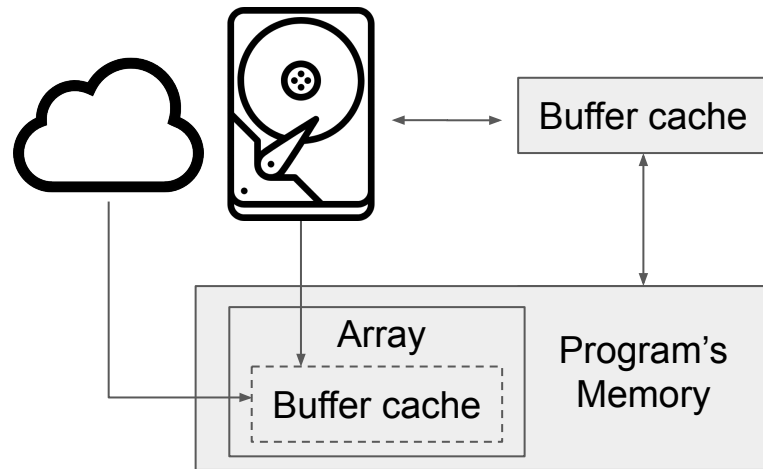
Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary

When Data > RAM

- Numpy Arrays

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





- Pandas Dataframes (v1.x)

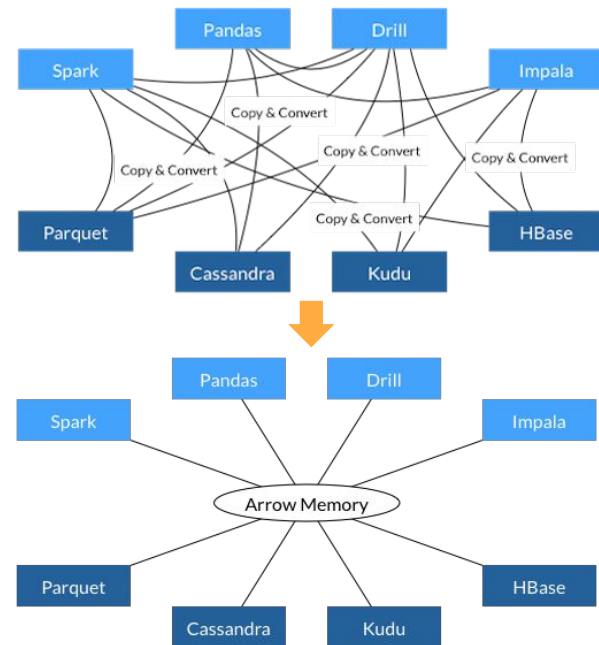
- Using Categorical dtypes and smaller subtypes
- Loading columns selectively: `pd.read_csv(file, usecols=[...])`
- Sampling rows: `pd.read_csv(file, skiprows=samplingFunc)`
- Reading in chunks: `for df in pd.read_csv(file, chunksize=1000):`
- Using SQLite as data storage for Pandas: 50x faster lookups

Arrow comes to the rescue!

- Apache Arrow: a competitor to Numpy
 - An open standard for processing and transporting large data
 - Using in-memory columnar format
 - Zero-copy data access

-  Polars: a superfast dataframe library
 - Written in Rust, internally using Arrow2 object
 - Support multi-threaded and SIMD
 - Lazy evaluation with query optimization and streaming data

-  **pandas v2.x**: published in March 2023
 - Add arrow as backend, perfect for data ETL
 - Internally using PyArrow object, slower than Polars, but with lighter cpu load
 - Numpy backend also got improved



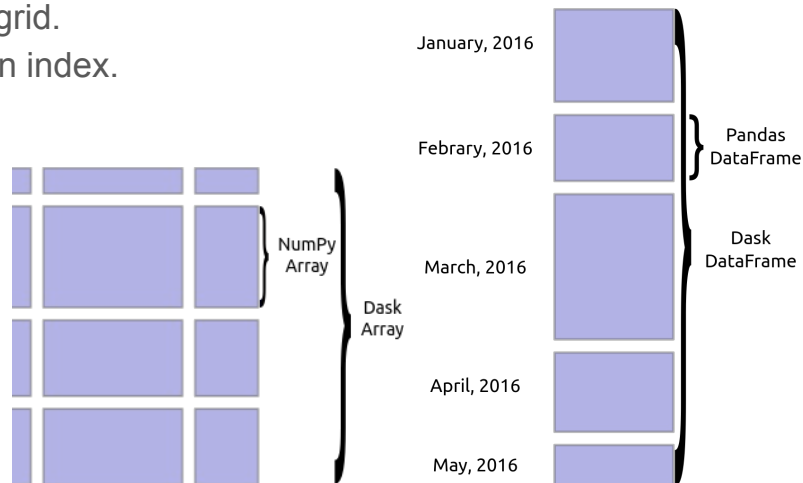
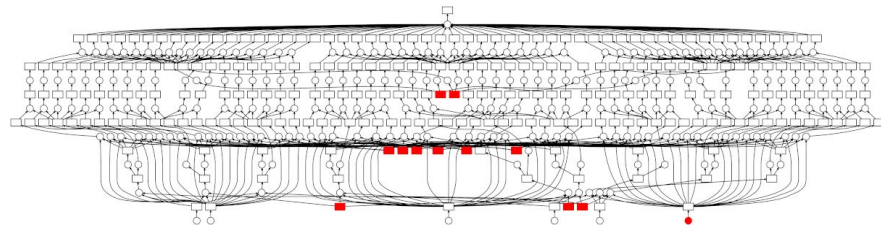


DASK Data Collections

bit.ly/hpdsby_03

- A full parallel processing library *suite*
 - Parallel with ndarray and pd.DataFrame
 - Distributed computing w/ task schedulers
 - 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()
```





- A promising open-source DataFrame library in Python
 - APIs closely resemble that of Pandas.
 - Easy to work with very large (>100G+) datasets efficiently, 1000x faster than Pandas!
- 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

Outline for today

- Key issues in array-type data
- Using single thread, single cpu
 - Numpy and Pandas 1.x
- Using multithread, multiple cpus
 - Pandarallel, Modin, Swifter
- When data is too large to fit the memory
 - Polars, Pandas 2.0, Dask, Vaex
- Summary

When to use which?

		Multiple CPUs	Out-Of-Core	Pandas APIs	Keep-In-Mind Notes
1	Pandas 2.0	✗	✗	✓	<ul style="list-style-type: none">• Always be the one if we can.• Lots of performance tricks!
2	Modin	✓	✗	✓	<ul style="list-style-type: none">• Good for dataframes with many columns• Out-of-core is linked with parallel engines• Experimental distributed XG-Boost
3	Dask	✓	✓	✓	<ul style="list-style-type: none">• Ultimate solution if distributed situations:<ul style="list-style-type: none">◦ Data storage & computation.• Steep learning curve after a quick start
4	Vaex	✓	✓	✗	<ul style="list-style-type: none">• Good for dataframes with many rows• HDF5 files work best so far.• Some APIs are different from Pandas

01

Use **Pandas** if possible

- Drop to numpy if you can.
- Try “dtype_diet” to save RAM for you.
- Use functions from well-established numpy-based libraries



02

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.



03

When to consider **Dask**

- Some machine learning tasks can get great performance boost in Dask.
- Your dataset has to be distributed stored and operated over a cluster.



04

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.



What we didn't talk about yet:

