DSC 102
Systems for Scalable Analytics

Haojian Jin

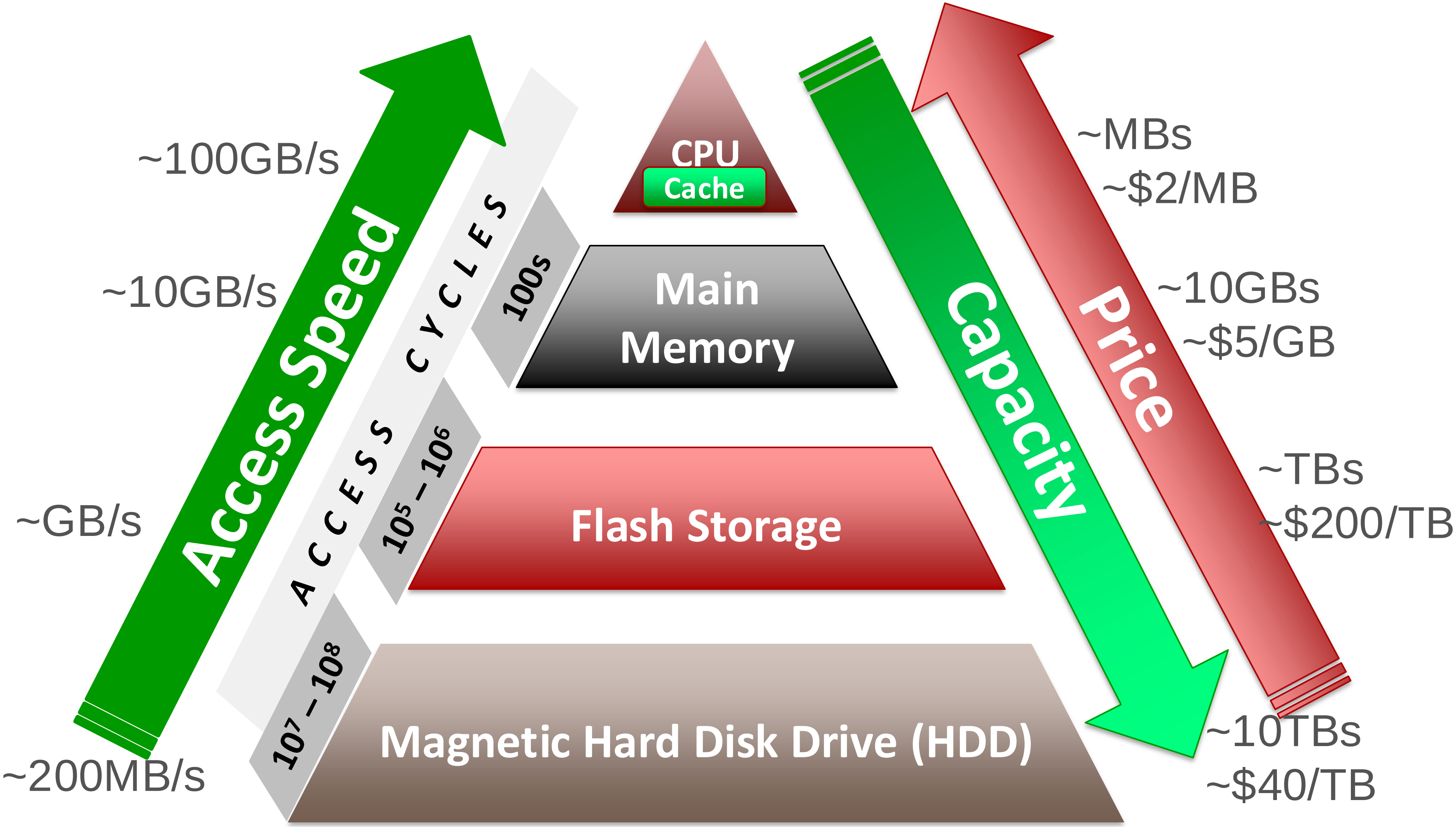
Topic 3: Parallel and Scalable Data Processing
Part 2: Scalable Data Access

Ch. 9.4, 12.2, 14.1.1, 14.6, 22.1-22.3, 22.4.1, 22.8 of Cow Book
Ch. 5, 6.1, 6.3, 6.4 of MLSys Book

Outline

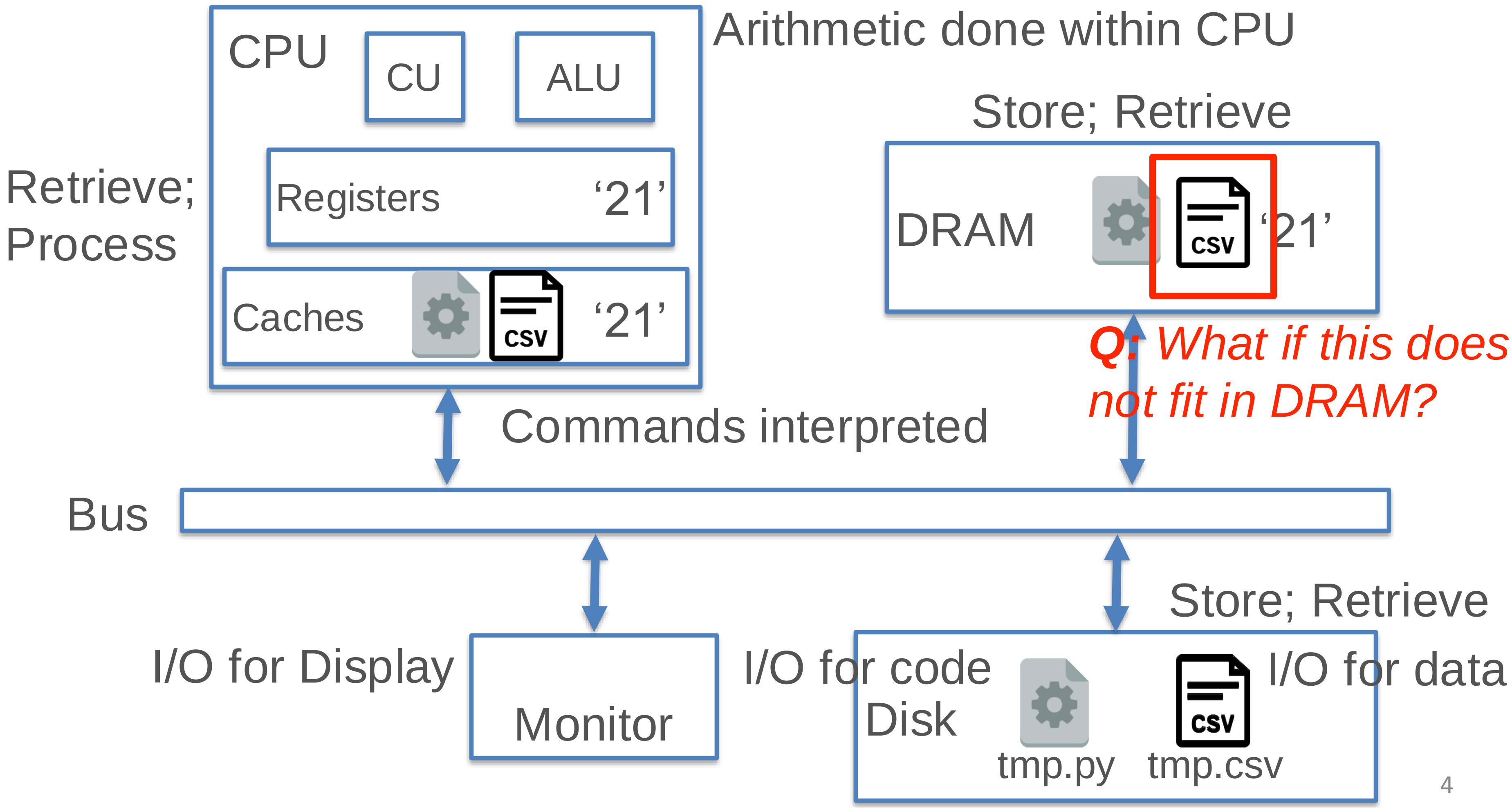
- Basics of Parallelism
  - Task Parallelism; Dask
  - Single-Node Multi-Core; SIMD; Accelerators
- Basics of Scalable Data Access
  - Paged Access; I/O Costs; Layouts/Access Patterns
  - Scaling Data Science Operations
- Data Parallelism: Parallelism + Scalability
  - Data-Parallel Data Science Operations
  - Optimizations and Hybrid Parallelism

Recap: Memory Hierarchy

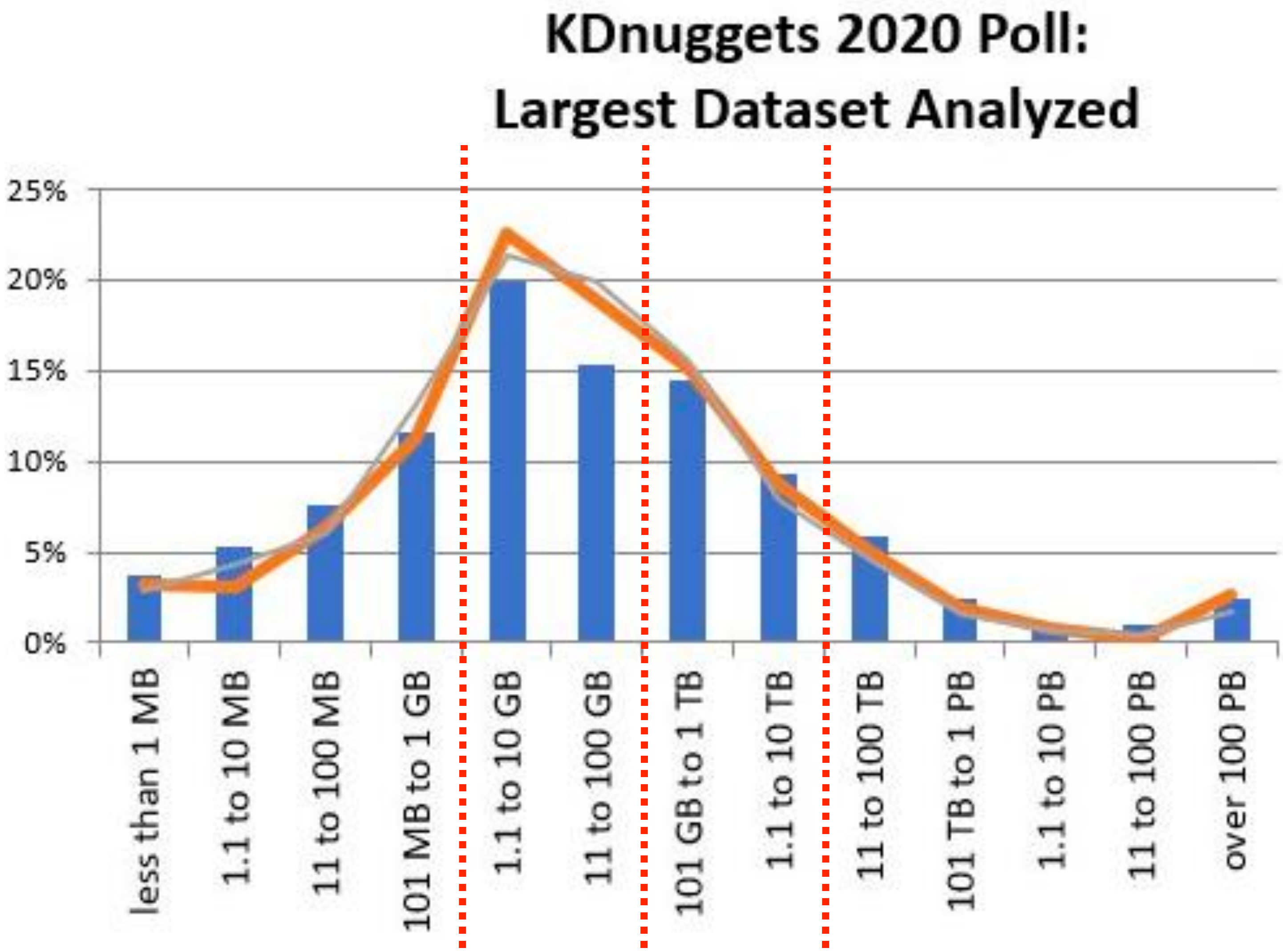


Memory Hierarchy in Action

Rough sequence of events when program is executed



Scale of Datasets in Practice



N = 562
62% industry

https://www.kdnuggets.com/2020/07/poll-largest-dataset-analyzed-results.html

Scalable Data Access

Central Issue: Large data file does not fit entirely in DRAM

Basic Idea: Divide-and-conquer again!
"Split" data file (virtually or physically) and ***stage reads*** of its pages from disk to DRAM; vice versa for writes

4 key regimes of scalability / staging reads:

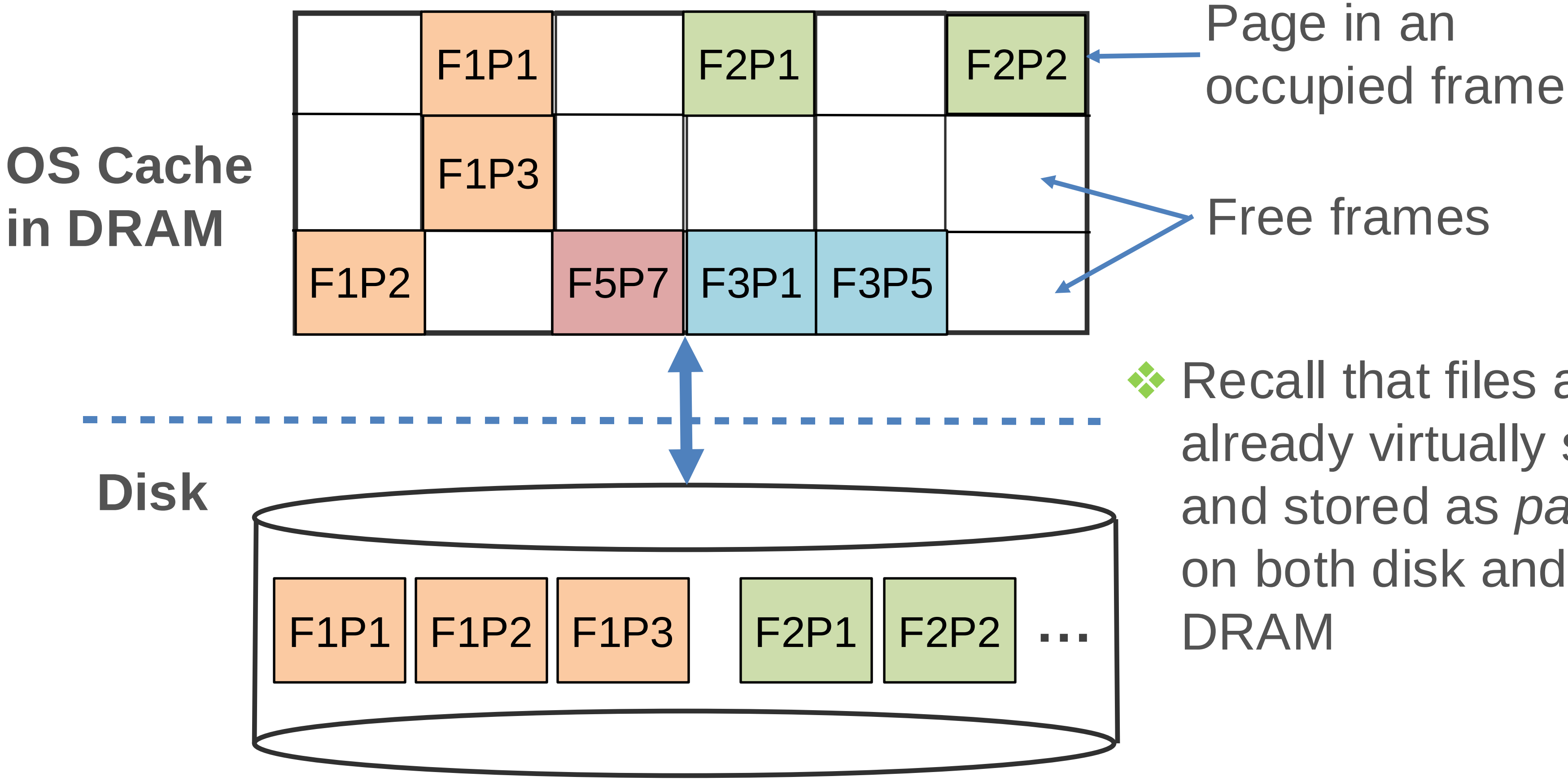
- **Single-node disk**: Paged access from file on local disk
- **Remote read**: Paged access from disk(s) over a network
- **Distributed memory**: Data fits on a cluster's total DRAM
- **Distributed disk**: Use entire memory hierarchy of cluster

Outline

- Basics of Parallelism
  - Task Parallelism; Dask
  - Single-Node Multi-Core; SIMD; Accelerators
- Basics of Scalable Data Access
  - Paged Access; I/O Costs; Layouts/Access Patterns
  - Scaling Data Science Operations
- Data Parallelism: Parallelism + Scalability
  - Data-Parallel Data Science Operations
  - Optimizations and Hybrid Parallelism

Paged Data Access to DRAM

Basic Idea: "Split" data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)



- Recall that files are already virtually split and stored as *pages* on both disk and DRAM

Page Management in DRAM Cache

- **Caching**: Retaining pages read from disk in DRAM
- **Eviction**: Removing a page frame's content in DRAM
- **Spilling**: Writing out pages from DRAM to disk
  - If a page in DRAM is "**dirty**" (i.e., some bytes were written), eviction requires a spill; o/w, ignore that page
- The set of DRAM-resident pages typically changes over the lifetime of a process
- **Cache Replacement Policy**: The algorithm that chooses which page frame(s) to evict when a new page has to be cached but the OS cache in DRAM is full
  - Popular policies include Least Recently Used, Most Recently Used, etc. (more shortly)

Quantifying I/O: Disk and Network

- Page reads/writes to/from DRAM from/to disk incur latency
- **Disk I/O Cost**: Abstract counting of number of page I/Os; can map to bytes given page size
- Sometimes, programs read/write data over network
- **Communication/Network I/O Cost**: Abstract counting of number of pages/bytes sent/received over network
- I/O cost is *abstract*; mapping to latency is *hardware-specific*

Example: Suppose a data file is 40GB; page size is 4KB
I/O cost to read file = 10 million page I/Os

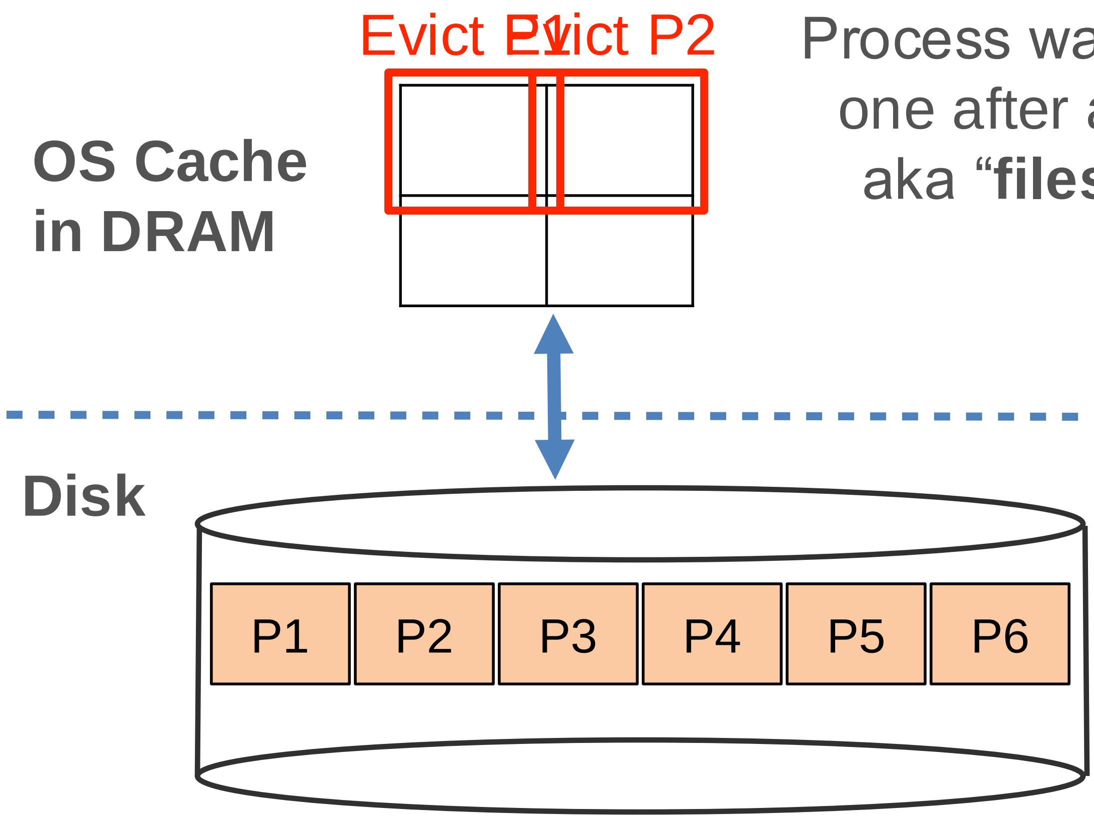
Disk with I/O throughput: 800 MB/s → 40GB/800MBps = 50s

Network with speed: 200 MB/s → 40GB/200MBps = 200s

Scaling to (Local) Disk

Basic Idea: Split data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)

Suppose OS Cache has only 4 frames; initially empty



Process wants to read file's pages one after another, then discard: aka "**filescan**" access pattern

Read P1
Read P2
Read P3
Read P4
Read P5
Read P6

Cache is full!

Cache Repl. needed

Total I/O cost: **6**

Scaling to (Local) Disk

- In general, *scalable programs stage access* to pages of file on disk and efficiently use available DRAM
  - Recall that typically DRAM size << Disk size
- Modern machines have 10s of GBs DRAM; so, read a "chunk"/"block" of file at a time (say, 1000s of pages)
  - On HDDs, such chunking leads to *more sequential I/Os*, raising throughput and lowering latency
  - Similarly, write a chunk of dirtied pages at a time

Data Layouts and Access Patterns

- **Data Layout:** Order in which data is laid out on storage; property of physical level of database
- **Data Access Pattern:** Order in which a program needs to access data for its computations; property of the program
- Together, the above two affect what data subset gets cached in higher level of memory hierarchy
- **Key Principle:** Optimizing data layout on disk based on data access pattern can help reduce I/O costs and latency
    - Applies to both HDDs and SSDs but especially critical for HDDs due to its random vs. sequential access latency gap

Row-store vs Column-store Layouts

- A common dichotomy when serializing 2-D structured data (relations, matrices, DataFrames) to file on disk

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

Say, a page can fit only 4 cell values

Row-store: | 1a,1b,1c,1d | 2a,2b,2c,2d | 3a,3b,3c,3d | …

Col-store: | 1a,2a,3a,4a | 5a,6a | 1b,2b,3b,4b | …

- Based on data access pattern of program, I/O costs with row- vs col-store can be orders of magnitude apart!

Row-store vs Column-store Layouts

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

Say, a page can fit only 4 cell values

Row-store: [1a,1b,1c,1d] [2a,2b,2c,2d] [3a,3b,3c,3d] …

Col-store: [1a,2a,3a,4a] [5a,6a] [1b,2b,3b,4b] …

Q: *What is the I/O cost with each to compute, say, a sum over B?*

- With row-store: need to fetch *all* pages; I/O cost: 6 pages
- With col-store: need to fetch only B's pages; I/O cost: 2 pages
- This difference generalizes to higher dim. for tensors

Hybrid/Tiled/“Blocked” Layouts

- Sometimes, it is beneficial to do a hybrid, especially for analytical RDBMSs and matrix/tensor processing systems

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

Say, a page can fit only 4 cell values

Hybrid stores with 2x2 tiled layout:

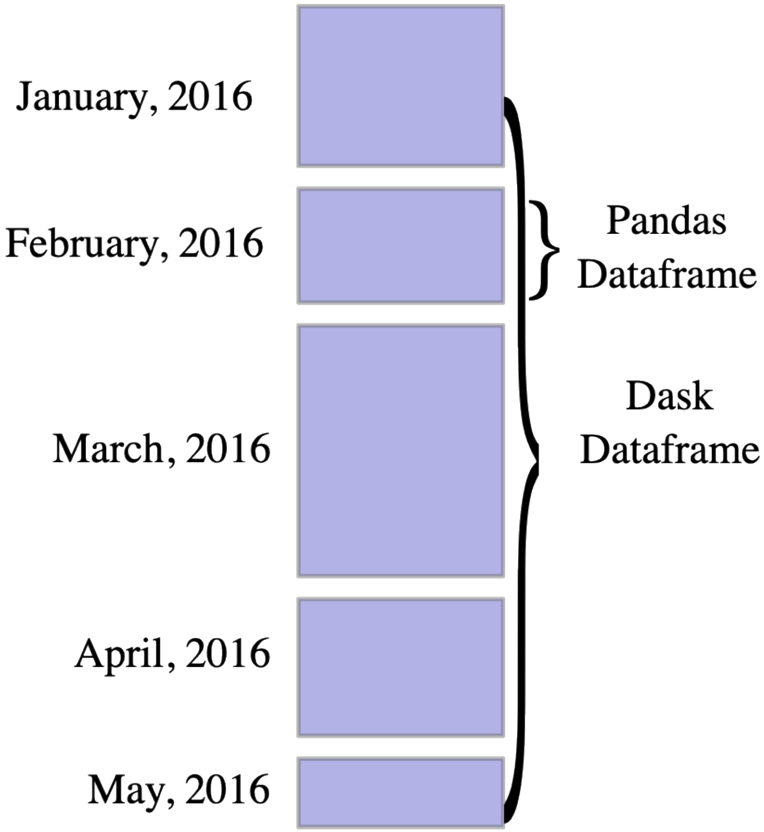
1a,1b,2a,2b	1c,1d,2c,2d	3a,3b,4a,4b	…
1a, 2a, 1b, 2b	1c, 2c, 1d, 2d	3a, 4a, 2b, 3b	…

Key Principle: Which data layout will yield lower I/O costs (row vs. col vs tiled) depends on data access pattern of the program!

Example: Dask's DataFrame

Basic Idea: Split data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)

- Dask DF scales to disk-resident data via a row-store
- "Virtual" split: each split is a Pandas DF under the hood
- Dask API is a "wrapper" around Pandas API to scale ops to splits and put all results together
- If file is too large for DRAM, need manual *repartition()* to get physically smaller splits (< ~1GB)

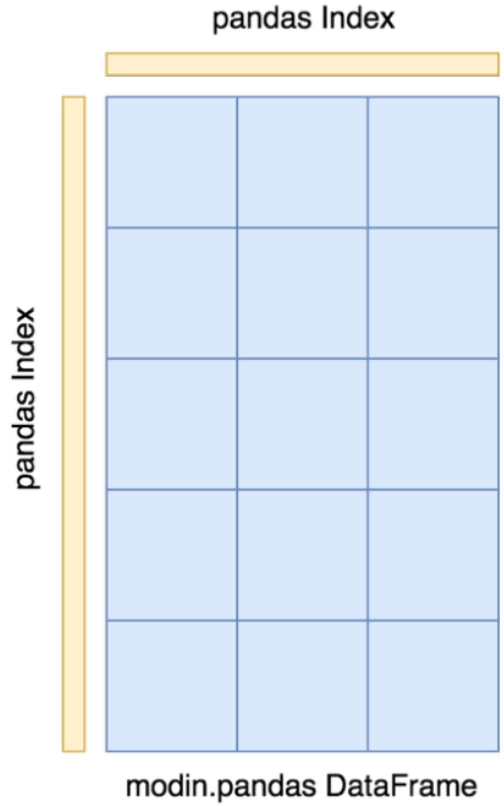


https://docs.dask.org/en/latest/dataframe-best-practices.html#repartition-to-reduce-overhead

Example: Modin's DataFrame

Basic Idea: Split data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)

- Modin's DF aims to scale to disk-resident data via a tiled store
- Enables seamless scaling along *both* dimensions
- Easier use of multi-core parallelism
- Many in-memory RDBMSs had this, e.g., SAP HANA, Oracle TimesTen
- ScaLAPACK had this for matrices



https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-191.pdf

Scaling with Remote Reads

Basic Idea: Split data file (virtually or physically) and *stage reads* of its pages from disk to DRAM (vice versa for writes)

- Similar to scaling to local disk but not “local”:
  - Stage page reads from remote disk/disks over the network (e.g., from S3)
- *More restrictive* than scaling with local disk, since spilling is not possible or requires costly network I/Os
  - OK for a *one-shot* filescan access pattern
  - Use DRAM to cache; repl. policies
  - Can also use smaller local disk as cache; you did this in PA1

Peer Instruction Activity

(Switch slides)

Outline

- Basics of Parallelism
  - Task Parallelism; Dask
  - Single-Node Multi-Core; SIMD; Accelerators
- Basics of Scalable Data Access
  - Paged Access; I/O Costs; Layouts/Access Patterns
  - Scaling Data Science Operations
- Data Parallelism: Parallelism + Scalability
  - Data-Parallel Data Science Operations
  - Optimizations and Hybrid Parallelism

Scaling Data Science Operations

- Scalable data access for key representative examples of programs/operations that are ubiquitous in data science:
  - DB systems:
    - Select
    - Non-deduplicating project
    - Simple SQL aggregates
    - GROUP BY aggregates
  - ML systems:
    - Matrix sum/norms
    - (Stochastic) Gradient Descent

Scaling to Disk: Non-dedup. Project

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

R

SELECT C FROM R

Row-store:

1a,1b,1c,1d	2a,2b,2c,2d	3a,3b,3c,3d
4a,4b,4c,4d	5a,5b,5c,5d	6a,6b,6c,6d

- Straightforward **filescan** data access pattern
  - Read one page at a time into DRAM; may need cache repl.
  - Drop unneeded columns from tuples on the fly
- I/O cost: 6 (read) + output # pages (write)

Scaling to Disk: Non-dedup. Project

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

R

SELECT C FROM R

Col-store:

1a,2a,3a,4a	5a,6a	1b,2b,3b,4b	5b,6b
1c,2c,3c,4c	5c,6c	1b,2b,3b,4b	5b,6b

- Since we only need col C, no need to read other pages
- I/O cost: 2 (read) + output # pages (write)
- Big advantage for col-stores over row-stores for SQL analytics queries (projects, aggregates, etc.), aka "OLAP"
  - Rationale for col-store RDBMS (e.g., Vertica) and Parquet

Scaling to Disk: Simple Aggregates

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

R

SELECT MAX(A) FROM R

Row-store:

1a,1b,1c,1d	2a,2b,2c,2d	3a,3b,3c,3d
4a,4b,4c,4d	5a,5b,5c,5d	6a,6b,6c,6d

- Again, straightforward **filescan** data access pattern
  - Similar I/O behavior as non-deduplicating project
- I/O cost: 6 (read) + output # pages (write)

Scaling to Disk: Simple Aggregates

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

R

SELECT MAX(A) FROM R

Col-store:

1a,2a,3a,4a	5a,6a	1b,2b,3b,4b	5b,6b
1c,2c,3c,4c	5c,6c	1b,2b,3b,4b	5b,6b

- Similar to the non-dedup. project, we only need col A; no need to read other pages!
- I/O cost: 2 (read) + output # pages (write)

Scaling to Disk: Group By Aggregate

A	B	C	D
a1	1b	1c	4
a2	2b	2c	3
a1	3b	3c	5
a3	4b	4c	1
a2	5b	5c	10
a1	6b	6c	8

R

SELECT A, SUM(D)
FROM R GROUP BY A

- Now it is not straightforward due to the GROUP BY!
- Need to "collect" all tuples in a group and apply agg. func. to each
- Typically done with a **hash table** maintained in DRAM
  - Has 1 record per group and maintains "running information" for that group's agg. func.
  - Built on the fly during filescan of R; holds the output in the end

Hash table (output)

A	Running Info.
a1	17
a2	13
a3	1

Scaling to Disk: Group By Aggregate

A	B	C	D
a1	1b	1c	4
a2	2b	2c	3
a1	3b	3c	5
a3	4b	4c	1
a2	5b	5c	10
a1	6b	6c	8

R

SELECT A, SUM(D)
FROM R GROUP BY A

Row-store:

a1,1b,1c,4	a2,2b,2c,3	a1,3b,3c,5
a3,4b,4c,1	a2,5b,5c,10	a1,6b,6c,8

Hash table in DRAM

A	Running Info.
a1	4 -> 9 -> 17
a2	3 -> 13
a3	1

- Note that the sum for each group is constructed *incrementally*
- I/O cost: 6 (read) + output # pages (write); just one filescan again!

***Q:** But what if hash table > DRAM size?!*

Scaling to Disk: Group By Aggregate

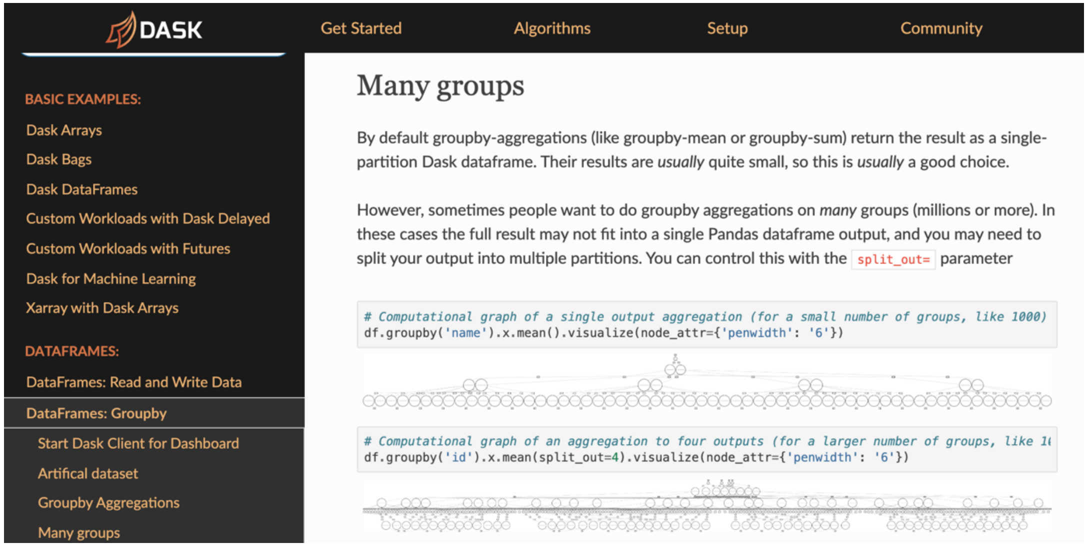
SELECT A, SUM(D) FROM R GROUP BY A

Q: *But what if hash table > DRAM size?*

- Program will likely just crash! OS may keep swapping pages of hash table to/from disk; aka "thrashing"

Q: *How to scale to large number of groups?*

- Divide and conquer! Split up R based on values of A
- HT for each split may fit in DRAM alone
- Reduce running info. size if possible



Ad: Take CSE 132C for more on how GROUP BY is scaled

Scaling to Disk: Relational Select

A	B	C	D
1a	1b	1c	1d
2a	2b	2c	2d
3a	3b	3c	3d
4a	4b	4c	4d
5a	5b	5c	5d
6a	6b	6c	6d

R

$$\sigma_{B=\text{“}3b\text{”}}(R)$$

SELECT C FROM R
WHERE B=“3b”

Row-store:

1a,1b,1c,1d	2a,2b,2c,2d	3a,3b,3c,3d
4a,4b,4c,4d	5a,5b,5c,5d	6a,6b,6c,6d

- Straightforward **filescan** data access pattern
  - Read pages/chunks from disk to DRAM one by one
  - CPU applies predicate to tuples in pages in DRAM
  - Copy satisfying tuples to temporary output pages
  - Use LRU for cache replacement, if needed
- I/O cost: 6 (read) + output # pages (write)

Scaling Data Science Operations

- Scalable data access for key representative examples of programs/operations that are ubiquitous in data science:
  - DB systems:
    - Select
    - Non-deduplicating project
    - Simple SQL aggregates
    - GROUP BY aggregates

Peer Instruction Activity

(Switch slides)

Review Questions

1. What are the 4 main regimes of scalable data access?
2. Briefly explain 1 pro and 1 con of scaling with local disk vs. scaling with remote reads.
3. You are given a DataFrame serialized as a 100 GB Parquet columnar file. It has 20 columns, all of the same fixed-length data type. You compute a sum over 4 columns. What is the I/O cost (in GB)?
4. Which is the most flexible data layout format for 2-D structured data?
5. You lay out a 1 TB matrix in tile format with a shape 2000x500. What is the I/O cost (in GB) of computing its full matrix sum?

Outline

- Basics of Parallelism
  - Task Parallelism; Dask
  - Single-Node Multi-Core; SIMD; Accelerators
- Basics of Scalable Data Access
  - Paged Access; I/O Costs; Layouts/Access Patterns
  - Scaling Data Science Operations
- Data Parallelism: Parallelism + Scalability
  - Data-Parallel Data Science Operations
  - Optimizations and Hybrid Parallelism