UC San Diego

DSC 102
Systems for Scalable Analytics

Arun Kumar

Topic 3: Parallel and Scalable Data Processing

Part 3: Data Parallelism

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

Introducing Data Parallelism

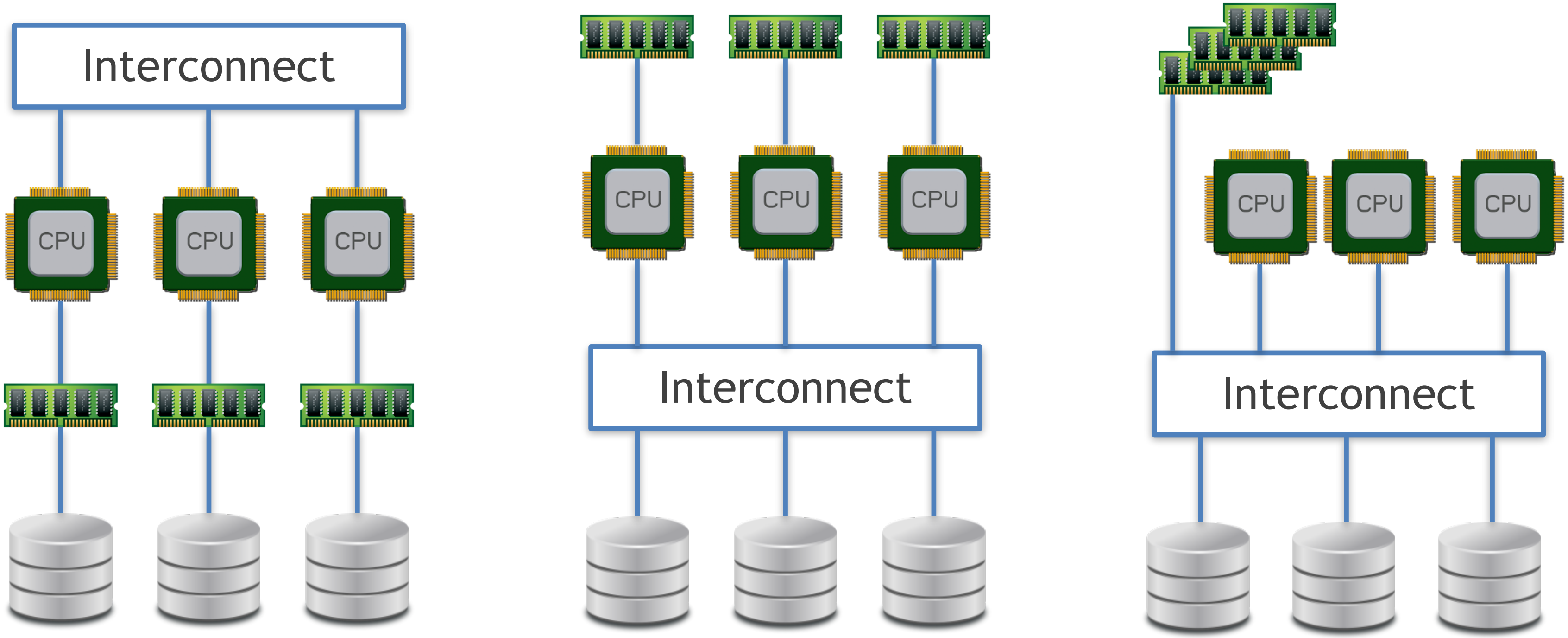
Basic Idea of Scalability: Split data file (virtually or physically) and *stage reads/writes* of its pages between disk and DRAM

***Q:** What is "data parallelism"?*

Data Parallelism: Partition large data file *physically* across nodes/workers; within worker: DRAM-based or disk-based

- The most common approach to marrying *parallelism* and *scalability* in data systems
- Generalization of SIMD and SPMD idea from parallel processors to large-scale data and multi-worker/multi-node setting
- Distributed-memory vs Distributed-disk

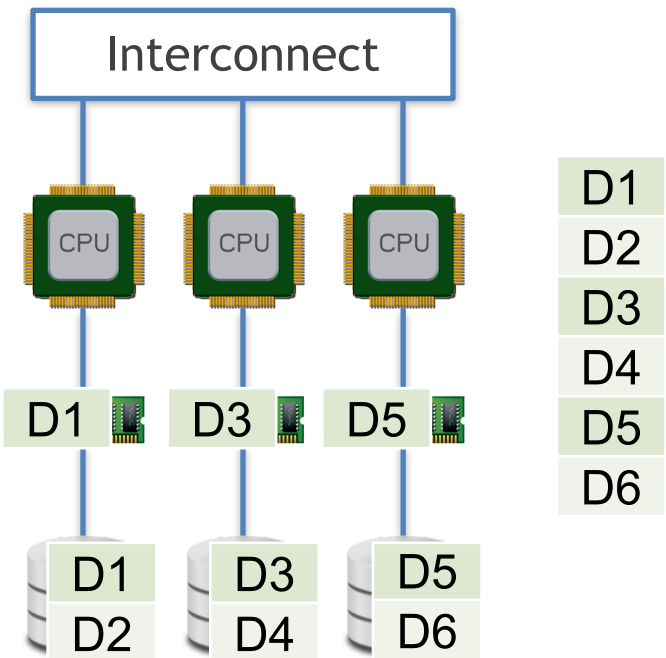
3 Paradigms of Multi-Node Parallelism



Shared-Nothing Parallelism | Shared-Disk Parallelism | Shared-Memory Parallelism

Data parallelism is technically *orthogonal* to these 3 paradigms but most commonly paired with shared-nothing

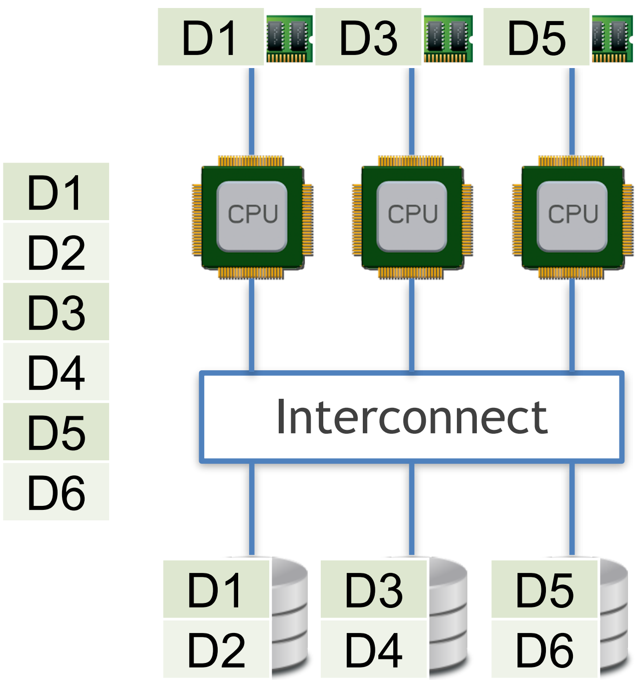
Shared-Nothing Data Parallelism



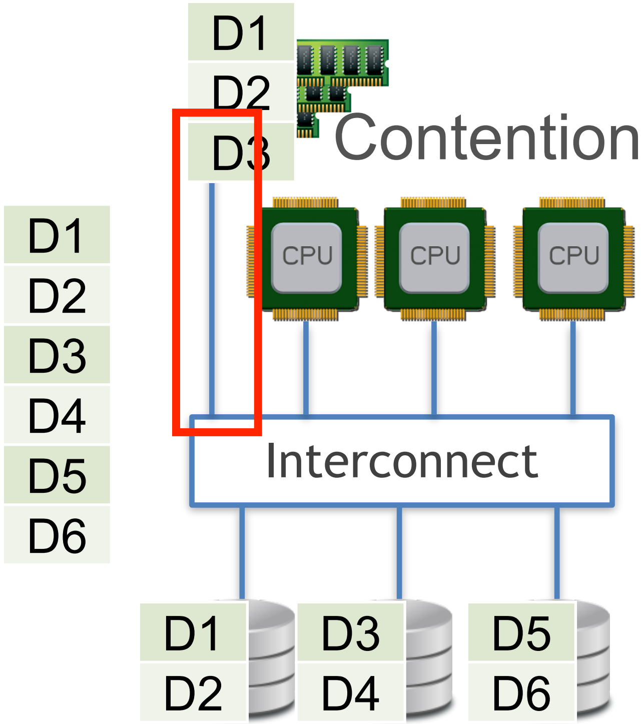
Shared-Nothing Parallel Cluster

- Partitioning a data file across nodes is aka **sharding**
- Part of a stage in data processing workflows called **Extract-Transform-Load** (ETL)
- ETL is an umbrella term for all kinds of processing done to the data file before it is ready for users to query, analyze, etc.
  - Sharding, compression, file format conversions, etc.

Data Parallelism in Other Paradigms?



Shared-Disk
Parallel Cluster



Shared-Memory
Parallel Cluster

Data Partitioning Strategies

- Row-wise/*horizontal* partitioning is most common (sharding)
- 3 common schemes (given k nodes):
  - **Round-robin**: assign tuple i to node i MOD k
  - **Hashing-based**: needs hash partitioning attribute(s)
  - **Range-based**: needs ordinal partitioning attribute(s)
- **Tradeoffs:**
  - Hashing-based most common in practice for RA/SQL
  - Range-based often good for range predicates in RA/SQL
  - But all 3 are often OK for many ML workloads (why?)
- **Replication** of partition across nodes (e.g., 3x) is common to enable *"fault tolerance"* and better parallel *runtime performance*

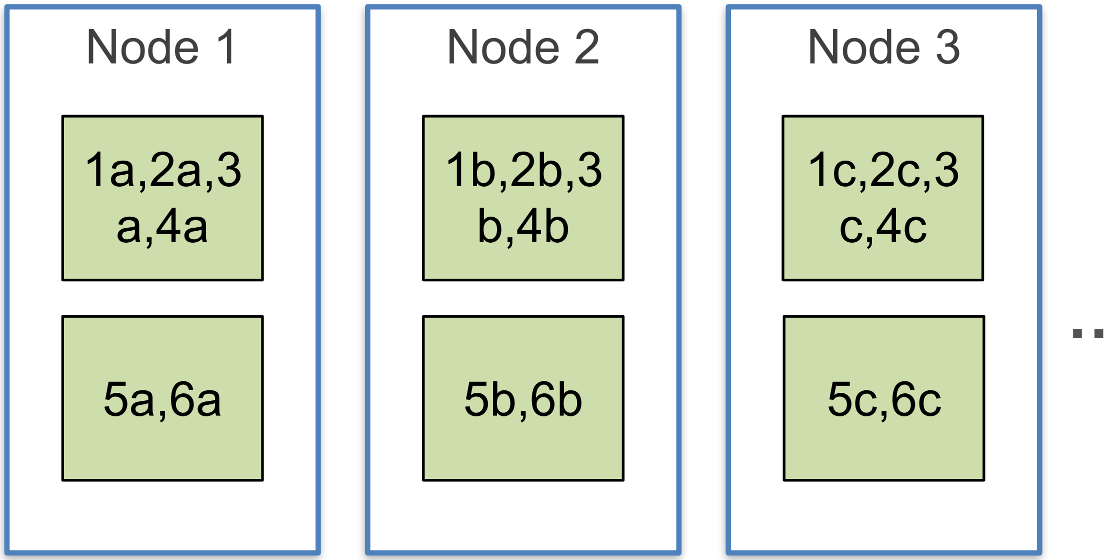
Other Forms of Data Partitioning

- Just like with disk-aware data layout on single-node, we can partition a large data file across workers in other ways too:

R

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

Columnar Partitioning



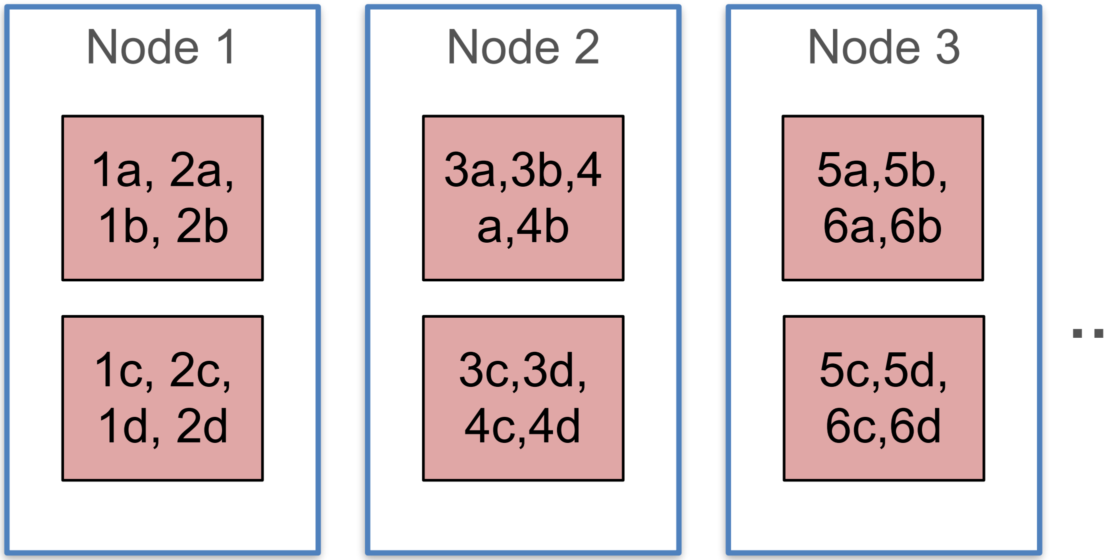
Other Forms of Data Partitioning

- Just like with disk-aware data layout on single-node, we can partition a large data file across workers in other ways too:

R

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

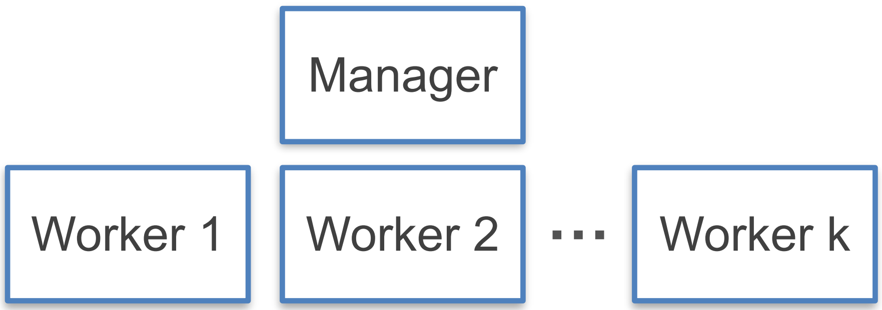
Hybrid/Tiled Partitioning



Cluster Architectures

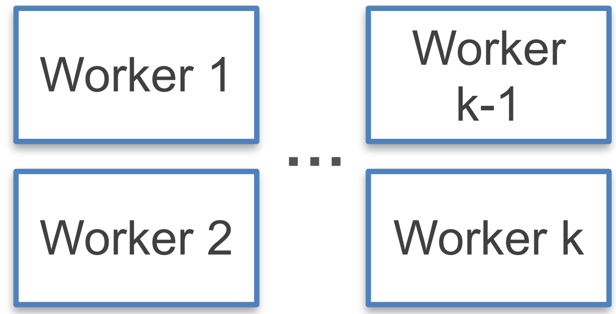
***Q:** What is the protocol for cluster nodes to talk to each other?*

Manager-Worker Architecture



- 1 (or few) special node called **Manager** (aka "Server" or archaic "Master"); 1 or more **Workers**
- Manager tells workers what to do and when to talk to other nodes
- Most common in data systems (e.g., Dask, Spark, par. RDBMS, etc.)

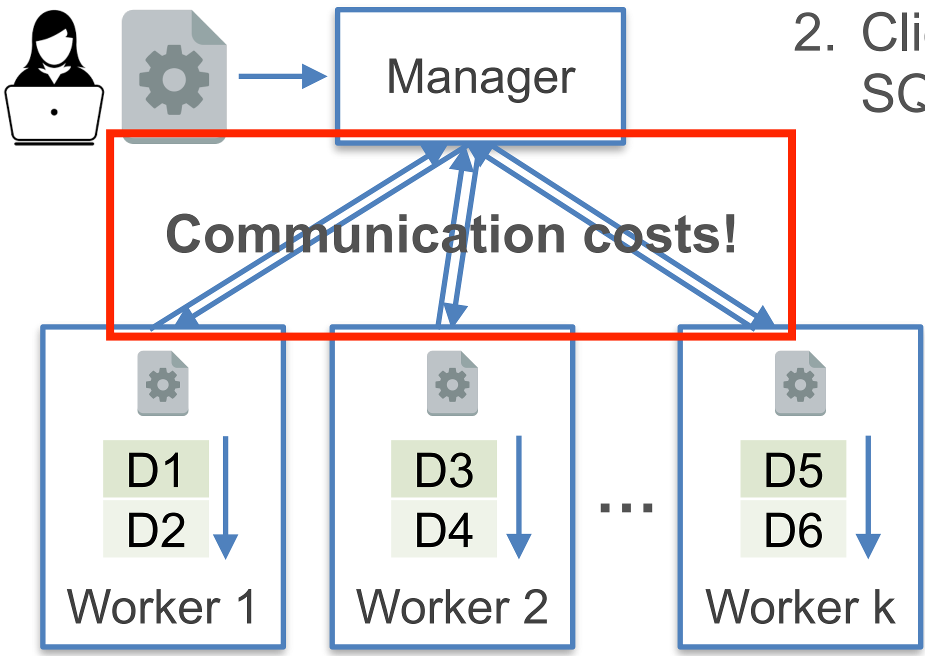
Peer-to-Peer Architecture



- No special manager
- Workers talk to each other directly
- E.g., Horovod
- Aka Decentralized (vs Centralized)

Bulk Synchronous Parallelism (BSP)

- Most common protocol of data parallelism in data systems (e.g., in parallel RDBMSs, Hadoop, Spark)
- Shared-nothing sharding + manager-worker architecture



Aka **(Barrier) Synchronization**

1. Sharded data file on workers
2. Client gives program to manager (e.g., SQL query, ML training, etc.)
3. Manager *divides* first piece of *work* among workers
4. Workers work *independently* on self's data partition (cross-talk can happen if Manager asks)
5. Worker sends partial results to Manager after one
6. Manager **waits** till all k done
7. Go to step 3 for next piece

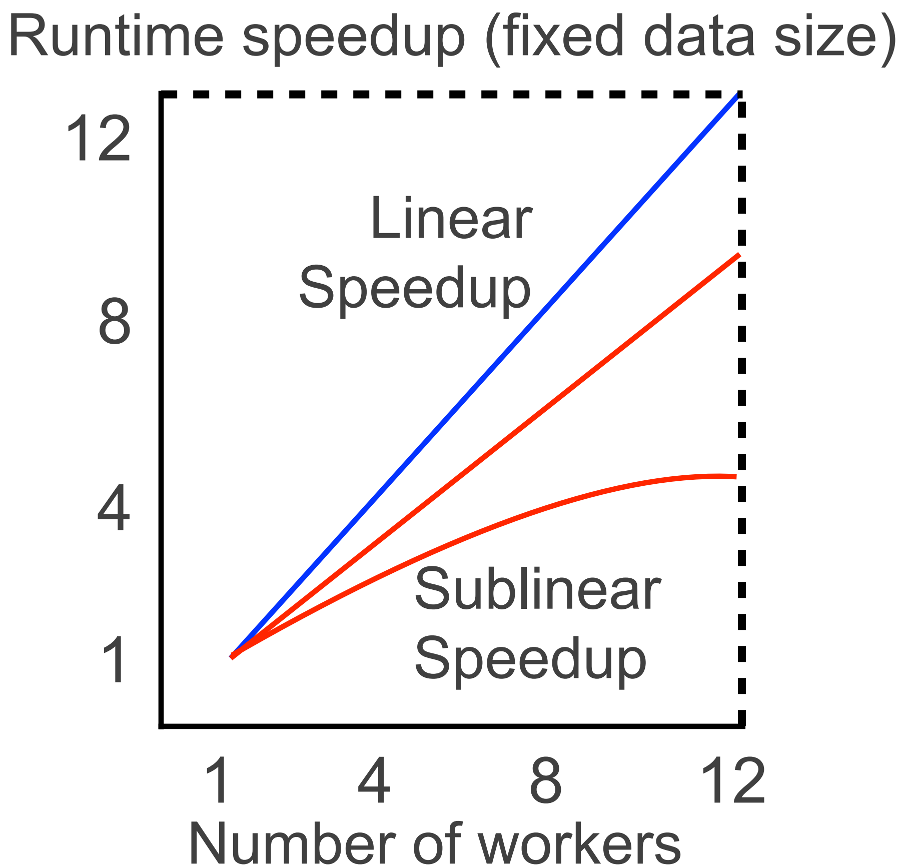
Speedup Analysis/Limits of of BSP

***Q:** What is the speedup yielded by BSP?*

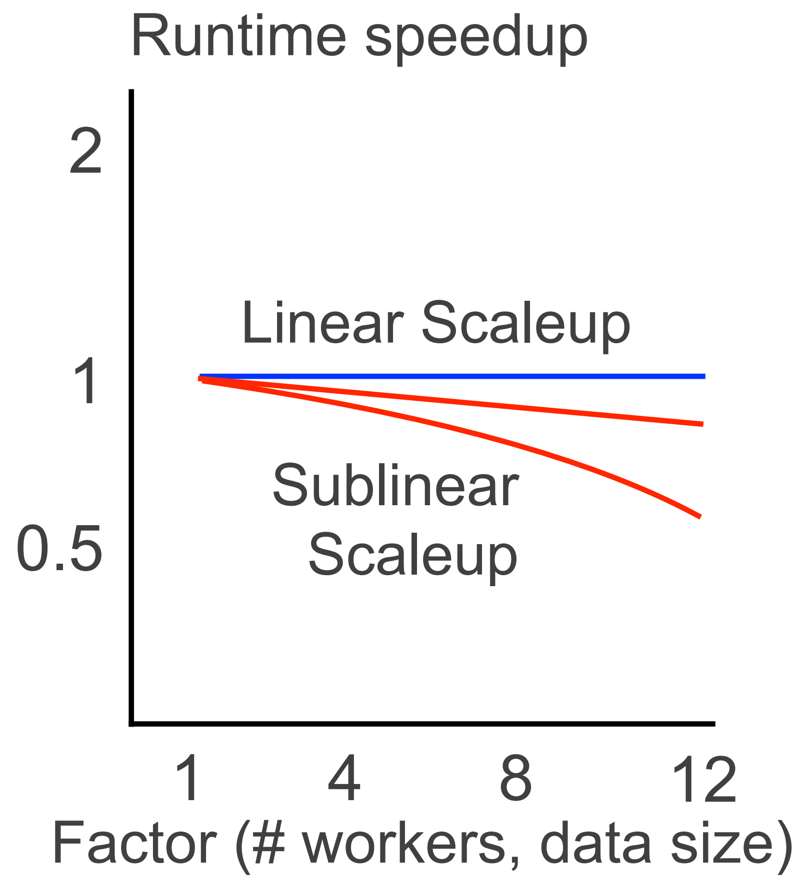
$$\textbf{Speedup} = \frac{\text{Completion time given only 1 worker}}{\text{Completion time given k (>1) workers}}$$

- Cluster overhead factors that hurt speedup:
  - **Per-worker:** startup cost; tear-down cost
  - **On manager:** dividing up the work; collecting/unifying partial partial results from workers
  - **Communication costs:** talk between manager-worker and across workers (when asked by manager)
  - Barrier synchronization suffers from "**stragglers**" due to skews in shard sizes and/or worker capacities

Quantifying Benefit of Parallelism



Speedup plot / Strong scaling



Scaleup plot / Weak scaling

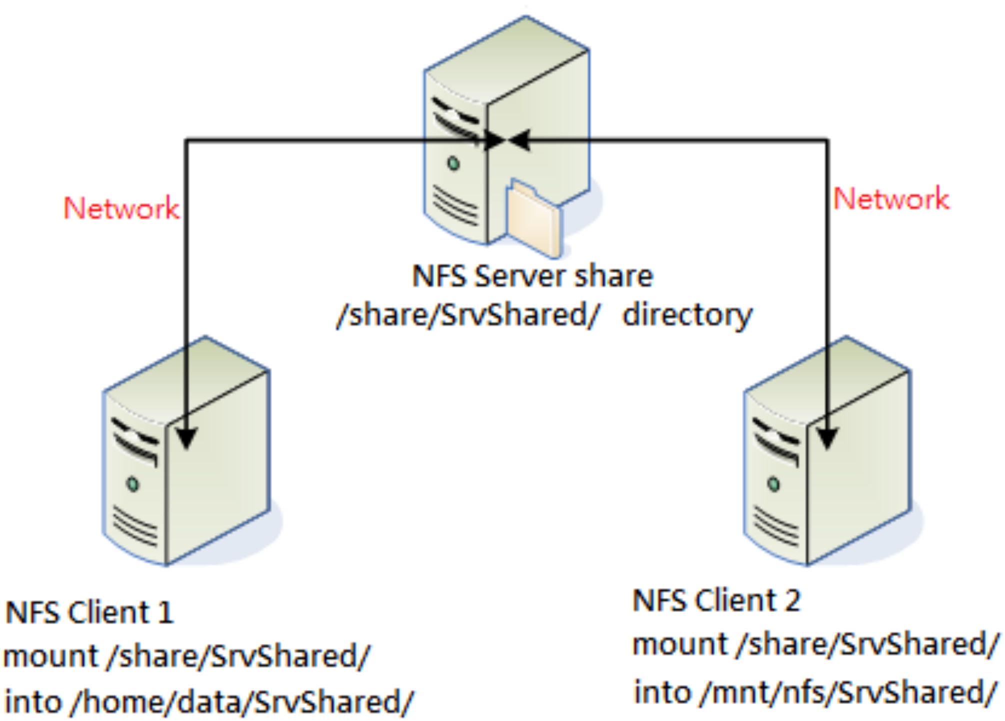
***Q:** Is <u>superlinear</u> speedup/scaleup ever possible?*

Distributed Filesystems

- Recall definition of file; *distributed file* generalizes it to a cluster of networked disks and OSs
- **Distributed filesystem (DFS)** is a cluster-resident filesystem to manage distributed files
  - A *layer of abstraction* on top of local filesystems
  - Nodes manage local data as if they are local files
  - *Illusion of a one global file*: DFS APIs let nodes access data sitting on other nodes
  - 2 main variants: Remote DFS vs In-Situ DFS
    - **Remote DFS:** Files reside elsewhere and read/written on demand by workers
    - **In-Situ DFS:** Files resides on cluster where workers exist

Network Filesystem (NFS)

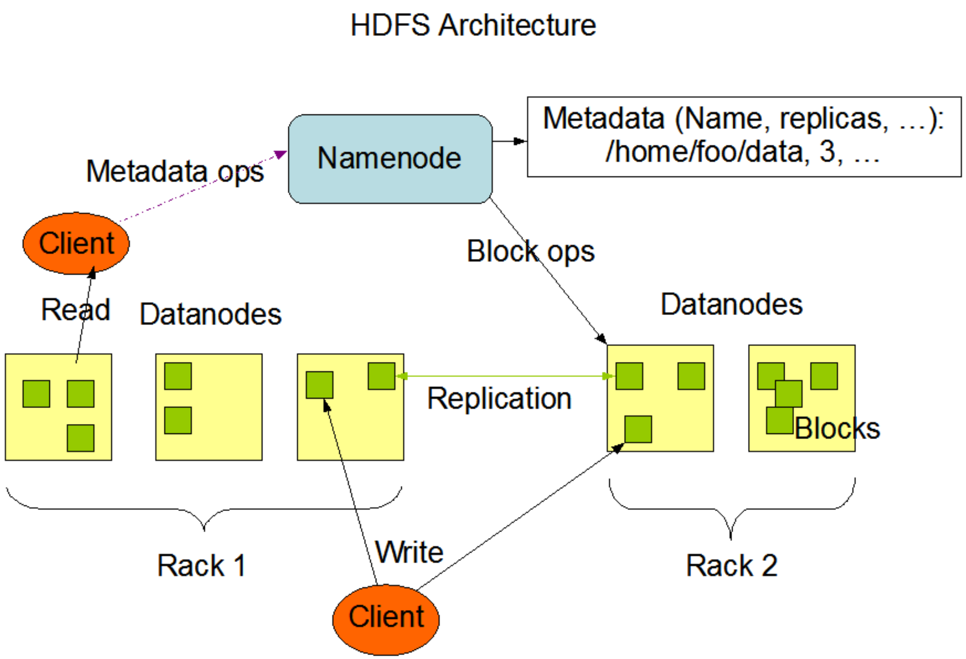
- An old remote DFS (c. 1980s) with simple client-server architecture for *replicating* files over the network



- Main pro: *simplicity* of setup and usage
- But many cons:
  - Not scalable to *very* large files
  - Full data replication
  - High contention for concurrent reads/writes
  - Single-point of failure

Hadoop Distributed File System (HDFS)

- Most popular in-situ DFS (c. late 2000s); part of Hadoop; open source spinoff of Google File system (GFS)
- *Highly scalable*; scales to 10s of 1000s of nodes, PB files



- Designed for clusters of cheap commodity nodes
- *Parallel* reads/writes of sharded data "blocks"
- Replication of blocks to improve *fault tolerance*
- Cons: Read-only + batch-append (no fine-grained updates/writes)

https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Hadoop Distributed File System (HDFS)

- NameNode's roster maps data blocks to DataNodes/IPs
- A distributed file on HDFS is just a directory (!) with individual filenames for each data block and metadata files

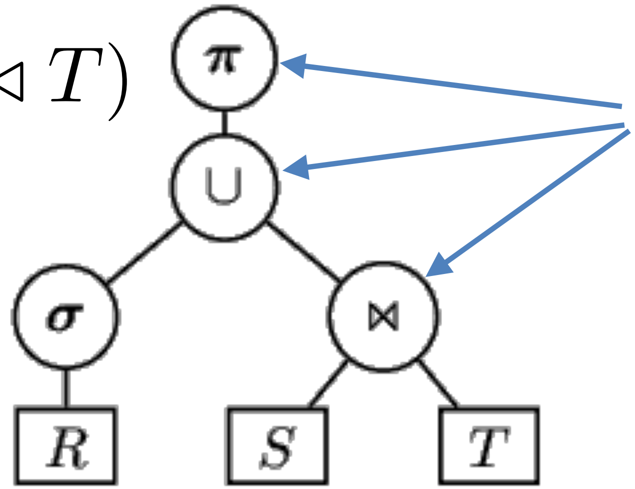
```
${dfs.datanode.data.dir}/
├ current
|  ├── BP-526805057-127.0.0.1-1411980876842
|  |   └── current
|  |    ├── VERSION
|  |    ├── finalized
|  |    |   ├──── blk_1073741825
|  |    |   ├──── blk_1073741825_1001.meta
|  |    |   ├──── blk_1073741826
|  |    |   └──── blk_1073741826_1002.meta
|  |    └── rbw
|  └── VERSION
└ in_use.lock
```

- HDFS *data block size* and *replication factor* are configurable parameters; default are 128 MB and 3x

Data-Parallel Dataflow/Workflow

- **Data-Parallel Dataflow:** A dataflow graph with ops wherein each operation is executed in a data-parallel manner
- **Data-Parallel Workflow:** A generalization; each vertex a whole task/process that is run in a data-parallel manner

$$\pi(\sigma(R) \cup S \bowtie T)$$



Each of these extended relational ops have scalable *data-parallel implementations* in parallel RDBMSs, Spark, etc.

All input tables are sharded

Q: So how do we run data sci. ops in data-parallel manner?

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

Data-Parallel Data Science Ops

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

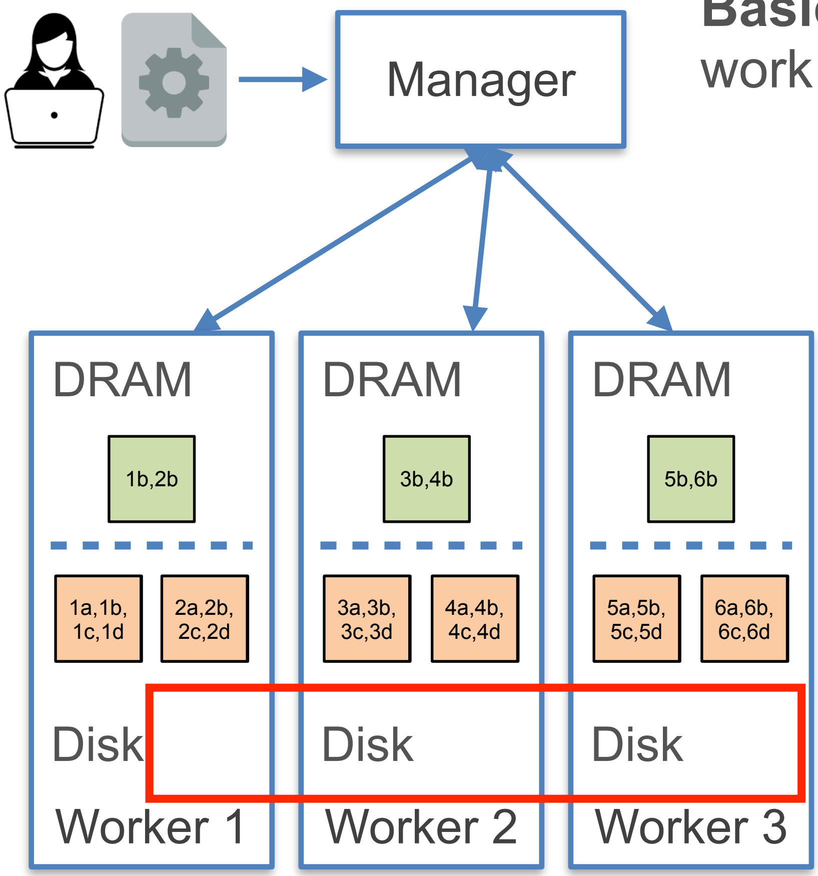
R

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

Data-Parallel Non-dedup. Project

SELECT C FROM R

We focus on BSP data-parallel



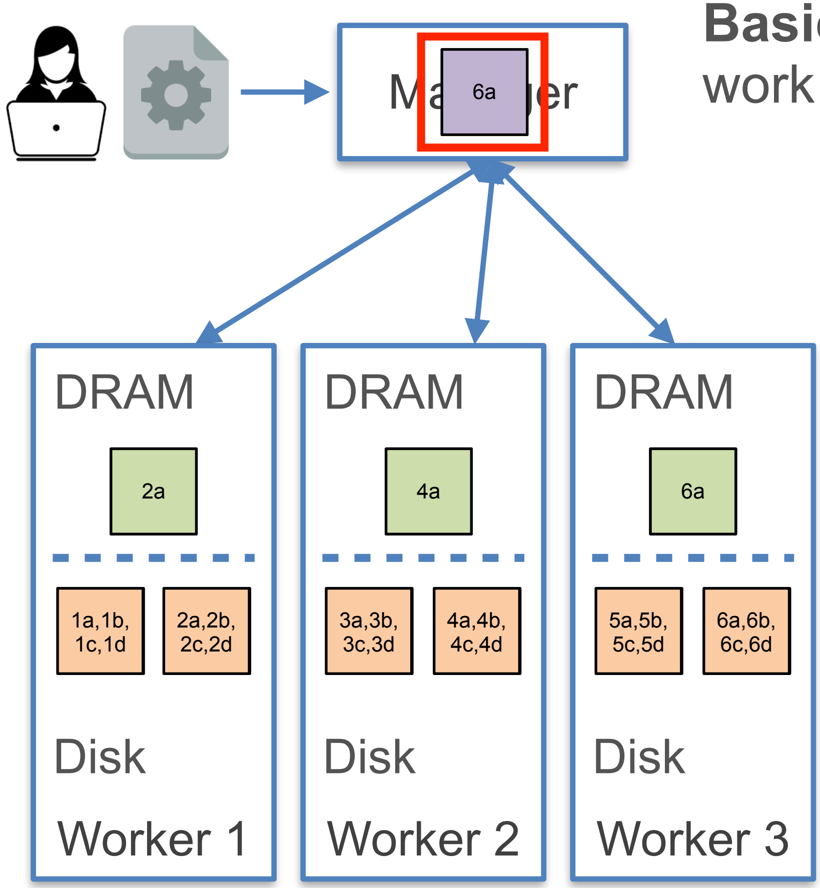
Basic Idea: Manager splits work -> node-local work -> manager unifies results

1. After ETL, sharded large input file sits cluster's disks
2. When query/program given, Manager broadcasts it as such
3. Each worker does node-local Non-dedup Project as explained before and writes local output to local file
4. Manager reports union of local files as global output file

I/O costs: Disk: 6 (pages) + output; Network: 0

Data-Parallel Simple Aggregates

SELECT MAX(A) FROM R



We focus on BSP data-parallel

Basic Idea: Manager splits work -> node-local work -> manager unifies results

1. After ETL, sharded large input file sits cluster's disks
2. When query/program given, Manager broadcasts it as such
3. Each worker does node-local simple **partial aggregate** as explained before and *sends it to Manager* for unification
4. Manager unifies partial results based on op semantics

I/O costs: Disk: 6 (pages) + output; Network: 3 (#workers)

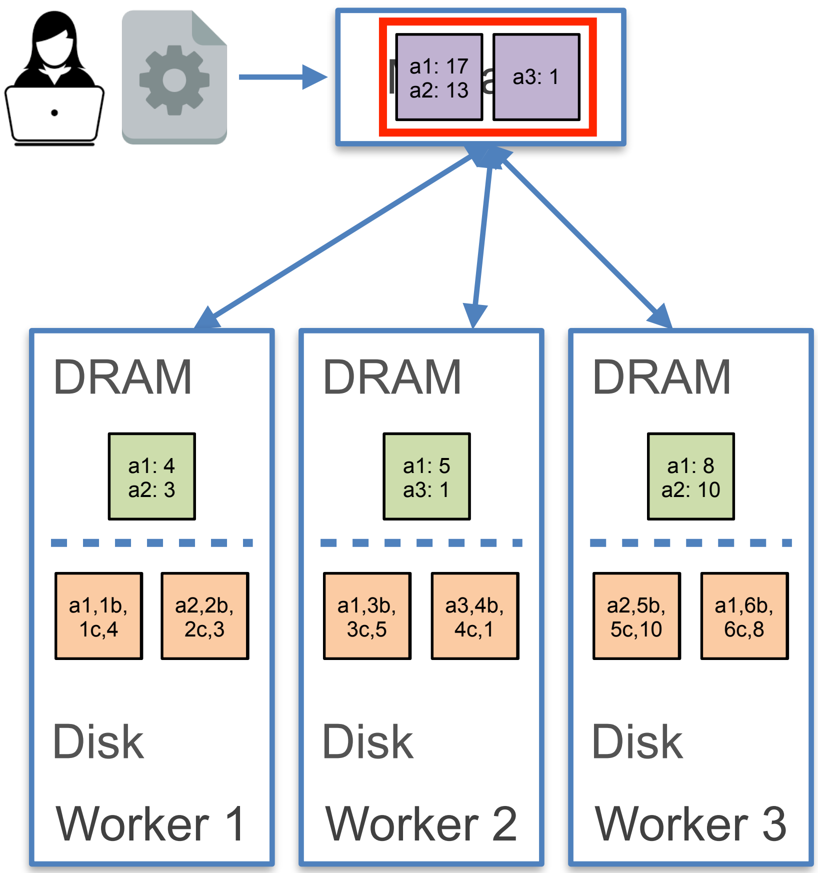
Data-Parallel Simple Aggregates

***Q:** Are all SQL aggregates easy to split up on sharded data?*

- Based on how easy it is to split up on shards, SQL aggs (aka descriptive stats) are categorized into 2/3 types:
- **Distributive Aggs:** A shard sends only 1 datum to manager
  - MIN, MAX, COUNT, SUM
- **Algebraic Aggs:** A shard sends O(1) size stats to manager
  - AVG (send SUM, COUNT separately); VARIANCE and STDEV (send SUM, SUM of squares, COUNT); etc.
- **Holistic Aggs:** Just O(1) size stats not enough in general; may need larger intermediate stats
  - MEDIAN, MODE, PERCENTILES, etc.

Data-Parallel Group By Aggregate

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



R

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

A	Running Info.
a1	17
a2	13
a3	1

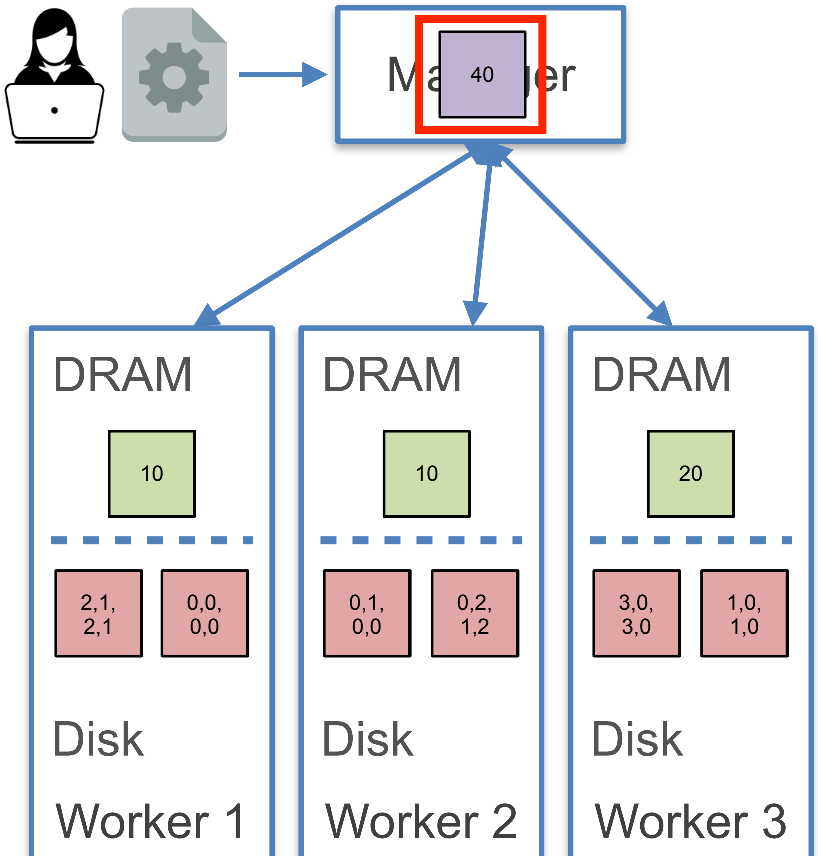
Output

- Similar to data-parallel simple agg
- Workers send **partial hash table** to manager based on local shards
- Manager collects and unifies local hash tables into global output
- Network I/O cost depends on data stats (domain size of A)

***Q:** What if Manager DRAM not enough to cache all hash tables?!*

Data-Parallel Matrix Sum/Norm

$$||M||_2^2$$



2	1	0	0
2	1	0	0
0	1	0	2
0	0	1	2
3	0	1	0
3	0	1	0

M_{6x4}

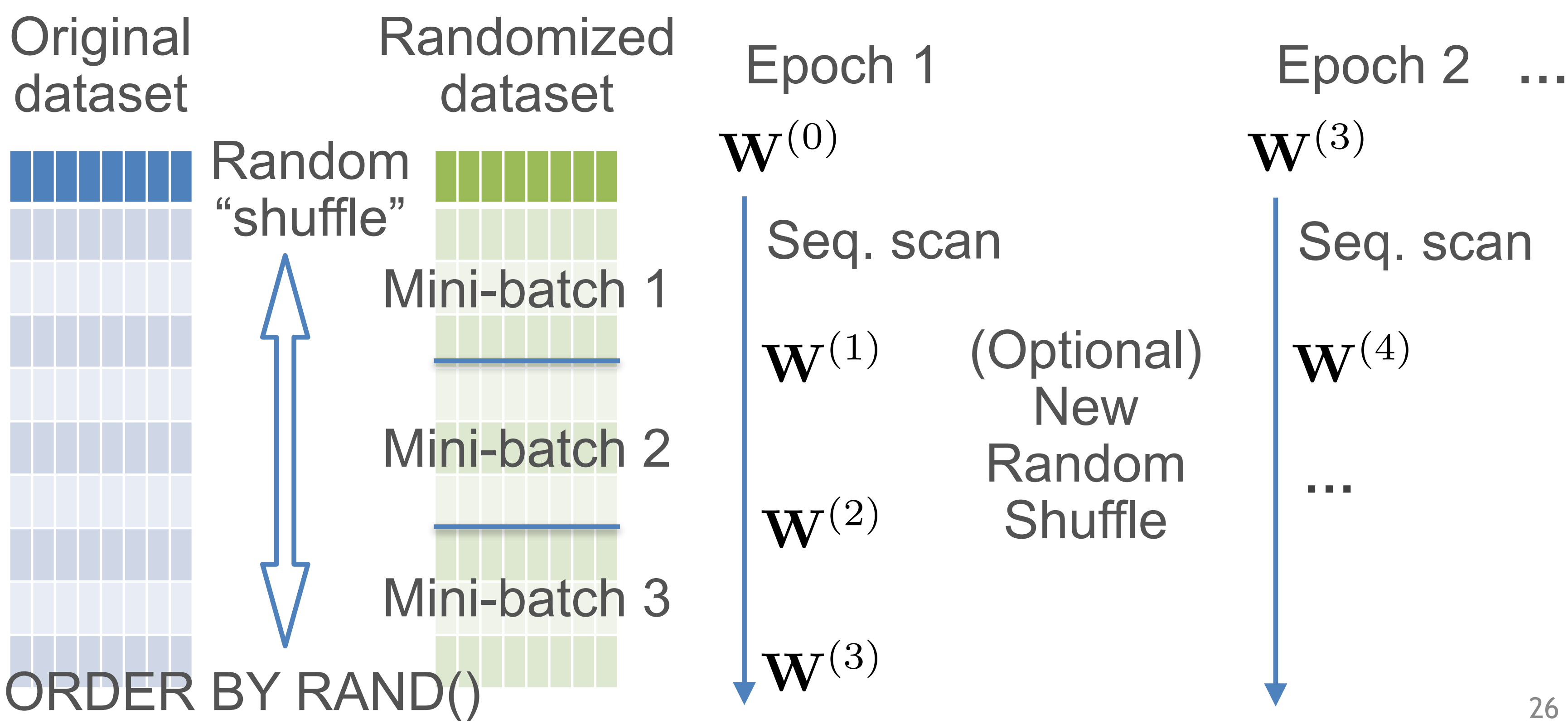
Say 2x2 tiled layout+partitioning

Similar to data-parallel simple agg
Disk I/O cost: 6 (pages)
Network I/O cost: 3 (#workers)

Data Access Pattern of Scalable SGD

$$\mathbf{W}^{(t+1)} \leftarrow \mathbf{W}^{(t)} - \eta \nabla \tilde{L}(\mathbf{W}^{(t)}) \qquad \nabla \tilde{L}(\mathbf{W}) = \sum_{i \in B} \nabla l(y_i, f(\mathbf{W}, x_i))$$

Sample mini-batch from dataset without replacement



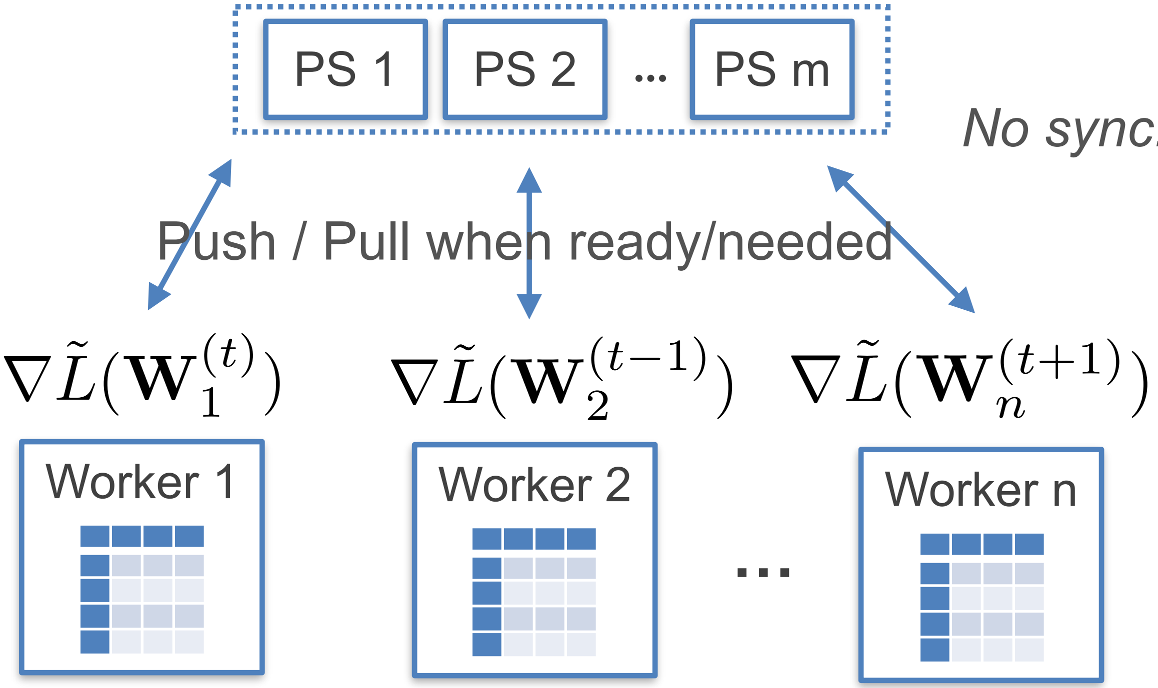
Data Access Pattern of Scalable SGD

- An SGD epoch is similar to SQL aggs but also different:
  - More complex agg. state (running info): model param. $\mathbf{W}^{(t)}$
  - Multiple mini-batch updates to model param. within a pass
  - Sequential dependency across mini-batches in a pass
  - Keep track of model param. across epochs
  - Not an *algebraic aggregate*; hard to parallelize!
  - Not *commutative*: different random shuffle orders give different results (very unlike relational ops)
  - (Optional) New random shuffling before each epoch

***Q:** How to execute SGD in a data-parallel manner?*

ParameterServer for Scalable SGD

Multi-server manager; each server manages a part of $\mathbf{W}^{(t)}$



No sync. for workers or servers

Workers send gradients to manager for updates at each mini-batch (or lower frequency)

Network I/O cost is high!

- Model params may get out-of-sync or stale; but SGD turns out to be robust; multiple updates/epoch helps

TensorFlow PyTorch

Ad: Take CSE 234 for more on parallel SGD and ML/DL systems

Data-Parallel Data Science Ops

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

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

Execution Optimization Tradeoffs

- Some common optimizations in data-parallel systems:
  - **Replication:** Put a shard on >1 worker; more parallelism possible for execution
  - **Caching:** Store as much data as possible on worker DRAM and/or disk
  - **Asynchrony:** Less common in DB systems; more common in ML systems (e.g., ParameterServer)
  - **Approximation:** Carefully exploit data subsampling
- Using ML for data placement, caching, tiered storage across memory hierarchy is now a hot topic in “ML for systems” world

Hybrid Parallelism

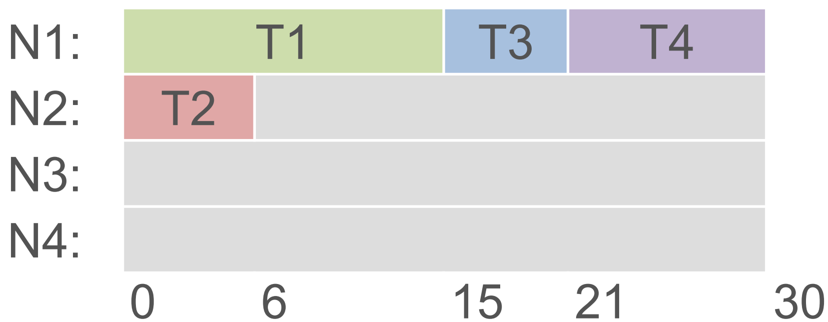
- Task- vs Data-Parallelism have pros and cons:
  - Task-par. wastes memory/storage due to replication; remote reads waste network; but easy to implement
  - Data-par. is painful to implement at op level; but scales w/o wasting memory/storage; more network costs

Q: *Is it possible to get the best of both these worlds?*

- Yes, often we can run task-par. on sharded data!
- Examples: Different SQL queries or different ML training routines run on top of same sharded data setup
  - Aka "**Multi-Query Execution**" in the DB world

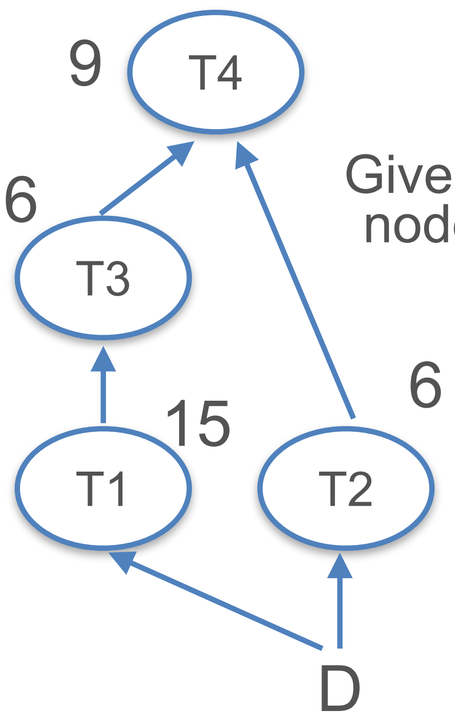
Task Par. vs BSP Data Par.

Fully task-par schedule:



N3, N4 are both useless. Why? N2 has idle times too. Why?

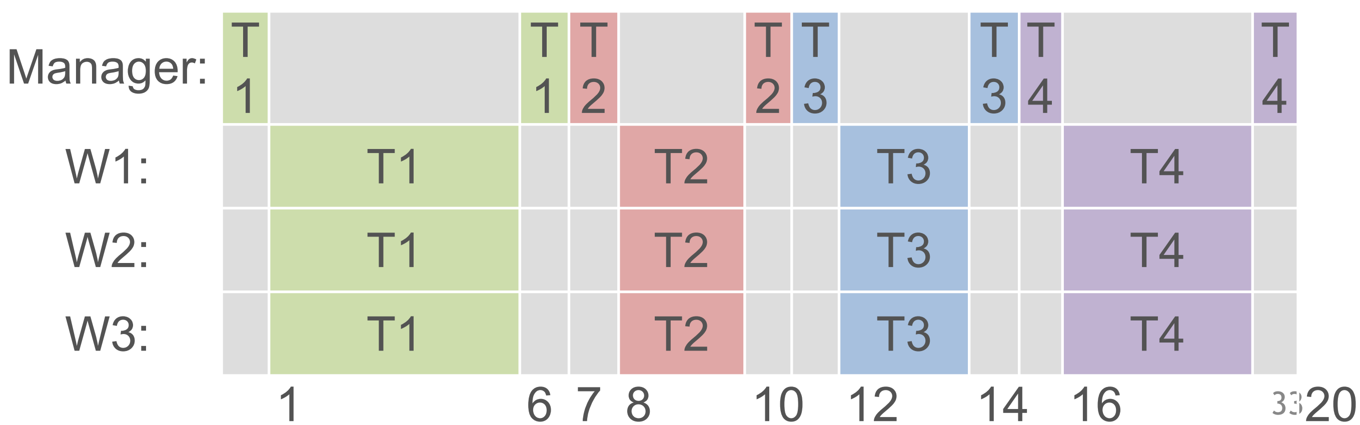
Example:



Given 4 nodes

Suppose each task gets perfect linear speedup on its useful work on BSP; manager overhead is, say, 1 unit each before/ after

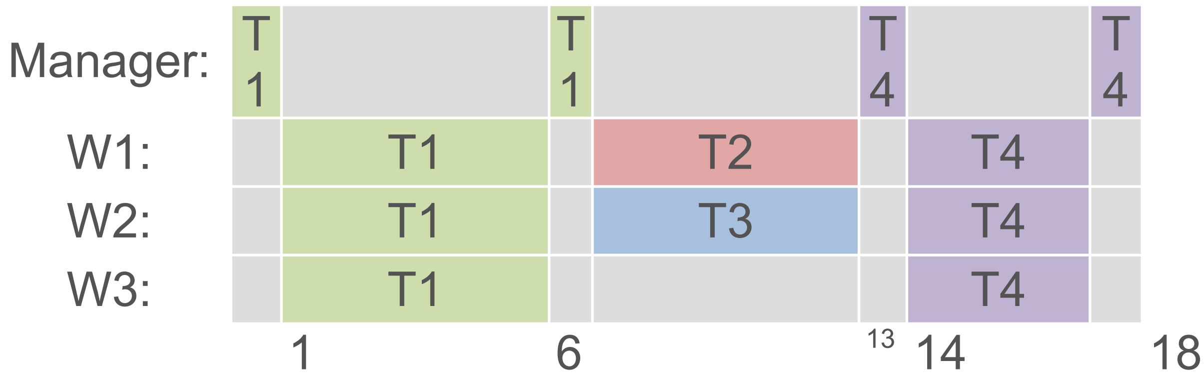
Fully BSP data-par schedule:



Hybrid of Task and Data Parallelism

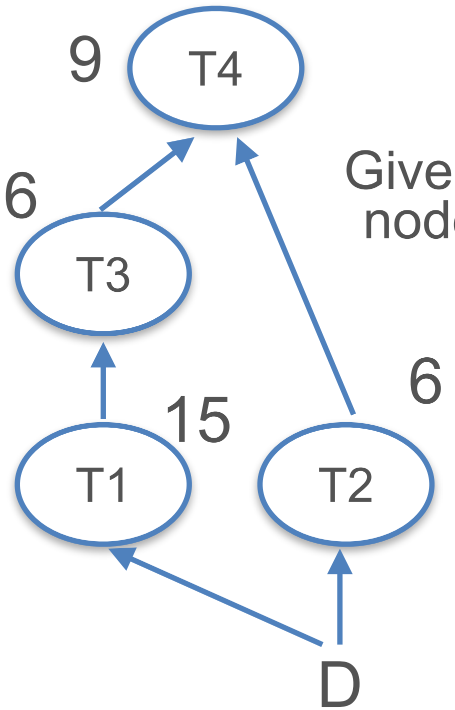
Q: *Can we go faster if we hybridize task and data par?*

One possible hybrid schedule:



Vs Fully task-par: 30 Vs Fully data-par: 20

Example:



Given 4 nodes

- Most scalable data systems today support only full task-par. (e.g., Dask) or full data-par. (e.g., RDBMS); hybrid software complexity is high
- Some RDBMSs do internally exploit hybrid-par. for relational dataflows
- Spark is beginning to support task-par. too

Hybrid of Task and Data Parallelism

***Q:** Can we go faster if we hybridize task and data par?*

- A key recent example from research: **Cerebro** for parallel DL model selection on clusters

Task-Parallel Systems | **Data-Parallel Systems**

No Partitioning (*Full replication*)	**Bulk** (*Shard level*)	**Fine-grained** (*Mini-batch level*)	
Dask, Celery, Vizier, Spark-HyperOpt, Ray	MOP/Cerebro (This Work)	Asynchronous Parameter Server	**Asynchronous**
	TensorFlow Model Averaging, Greenplum	Horovod, Synchronous Parameter Server	**Synchronous**

- First known form of "Bulk *Asynchronous* Parallelism"
- Resource-optimal when compute, memory/ storage, and network considered holistically

https://adalabucsd.github.io/cerebro.html (Start with the CIDR'21 paper and talk video)

Ad: Take CSE 234 for more on Cerebro, model selection systems

Review Discussion

1. To which multi-node parallelism paradigm (Shared Nothing/Memory/Disk) does data parallelism apply?
2. What are the two most common types of cluster communication protocols in parallel data systems?
3. Is it possible to pair up columnar partitioning with row store? Vice versa?
4. What exactly is the “synchrony” in BSP?
5. Name 2 common sources of overhead in data-parallel systems that can lead to sub-linear speedups.
6. Name 2 SQL aggregates that are NOT algebraic.
7. Why is SGD not amenable to parallelization like algebraic aggregates?
8. Why does Parameter Server have high communication costs when executing data-parallel SGD on a cluster?
9. Briefly explain 2 systems-level optimizations in data-parallel systems and how they can benefit data science workloads.
10. Name 1 pro and 1 con of BSP over task parallelism. Why do most parallel data systems today employ only one or the other?

Optional: More Examples of Data-Parallel Data Science Operations

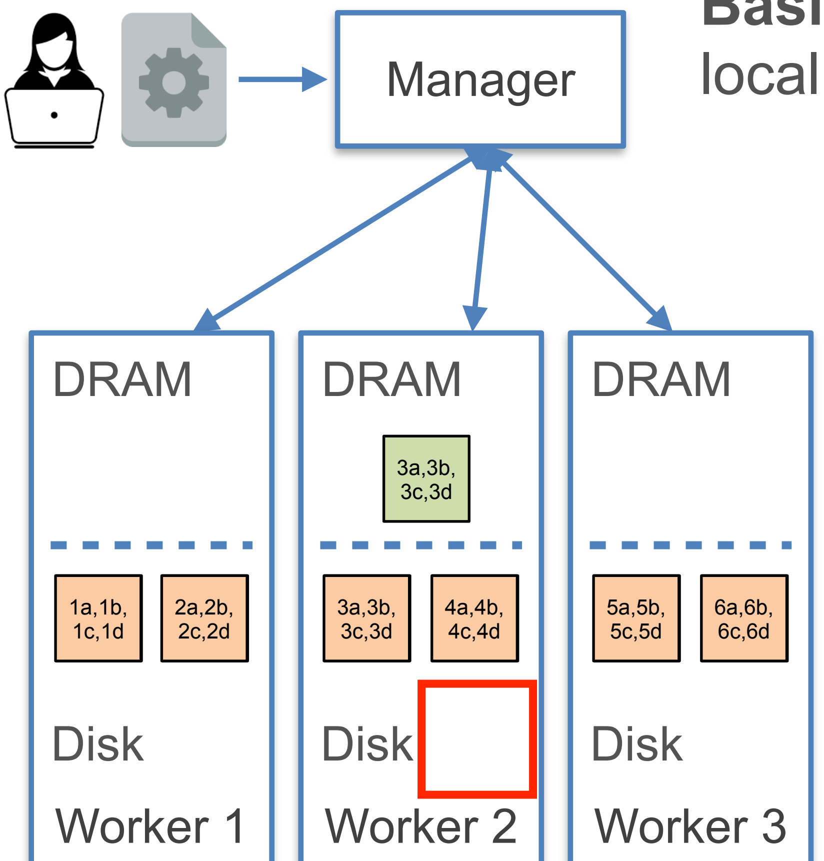
Not included in syllabus

Data-Parallel Relational Select

$$\sigma_{B=\text{“}3b\text{”}}(R)$$

We focus on BSP data-parallel

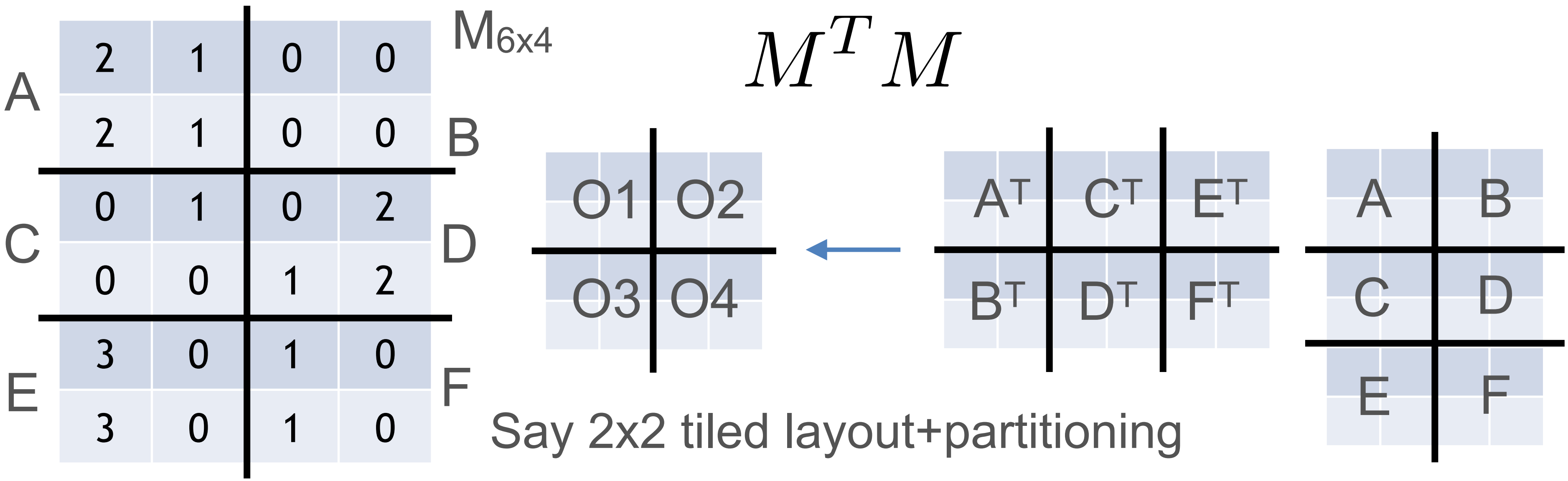
Basic Idea: Manager splits work -> node-local work -> manager unifies results



1. After ETL, sharded large input file sits cluster’s disks
2. When query/program given, manager broadcasts it as such
3. Each worker does node-local Select as explained before and writes local output to local file
4. Manager reports union of local files as global output file; note that output is also sharded file!

I/O costs: Disk: 6 (pages) + output; Network: 0

Data-Parallel Gramian Matrix

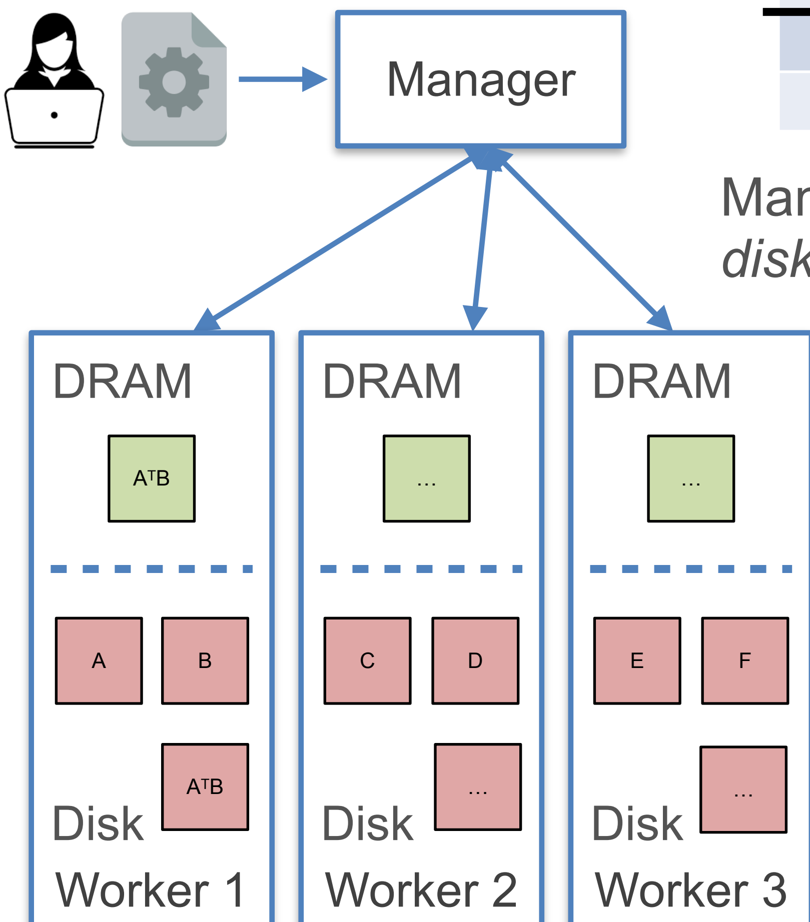


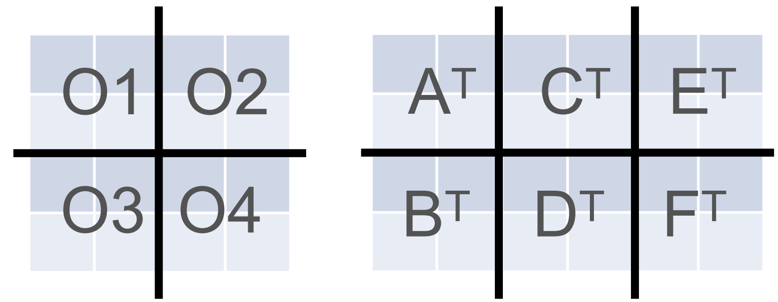
More complex in the data-parallel setting, since we may need to *communicate data shards* across workers!

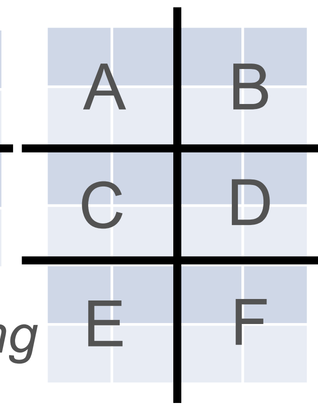
Basic Idea: Manager splits work -> node-local work -> *manager commands workers to talk to others* as needed -> more node-local work -> manager unifies results

Data-Parallel Gramian Matrix

$$M^T M$$







Many ways to do this, with *differing disk and network I/O costs*!

1. Manager *assigns tiles of* O to workers, e.g., W1: O1; W2: O2; W3: O3,O4
2. Workers compute node-local partial tiles needed and cache them on local disk: W1: A^TA, B^TB, A^TB, etc.
3. Manager tells workers to send corresp. tiles to other workers based on 2: W1 gets C^TC from W2 and E^TE from W3 for O1; W2 gets A^TB from W1; etc.
4. More node-local work to finish all O_i
5. Union of local tiles is sharded output!

Data-Parallel Gramian Matrix

- Not straightforward to determine I/O costs (both disk I/O and network I/O) of matrix mult., even simple Gramian!
  - CPU costs can also differ based on whether workers repeat redundant work vs cache it to file
  - Runtime is a complex function combining disk I/O cost, network I/O cost, and CPU/compute cost
- Different **operator implementations** exist in the parallel data systems literature: crossproduct-based multiply, replication-based multiply, etc.

https://sfu-db.github.io/dbsystems/Papers/systemML.pdf
http://www.vldb.org/pvldb/vol9/p1425-boehm.pdf