UC San Diego

DSC 102
Systems for Scalable Analytics

Rod Albuyeh

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

Scaling Data Science Operations

- Scalable data access is used in key representative examples of programs/operations that are ubiquitous in data science:
  - DB systems:
    - Non-deduplicating (as in, not avoiding duplication) project
    - Simple SQL aggregates (min, max, sum, etc)
    - SQL GROUP BY aggregates (agg group by key)
  - ML systems:
    - Matrix Sum of Squares
    - (Stochastic) Gradient Descent

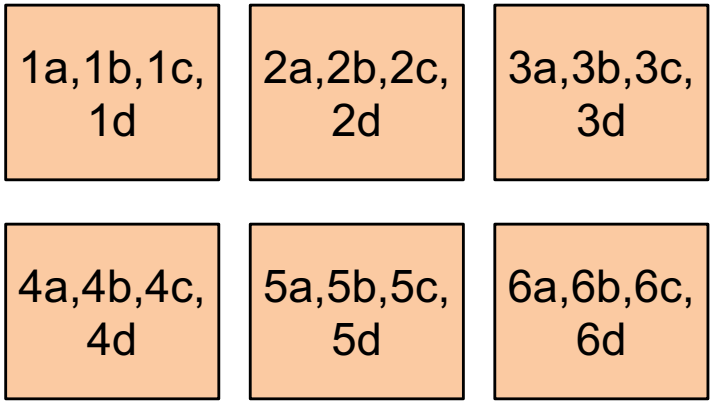
Scaling to Disk: Non-dedup. Project

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:



- 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)

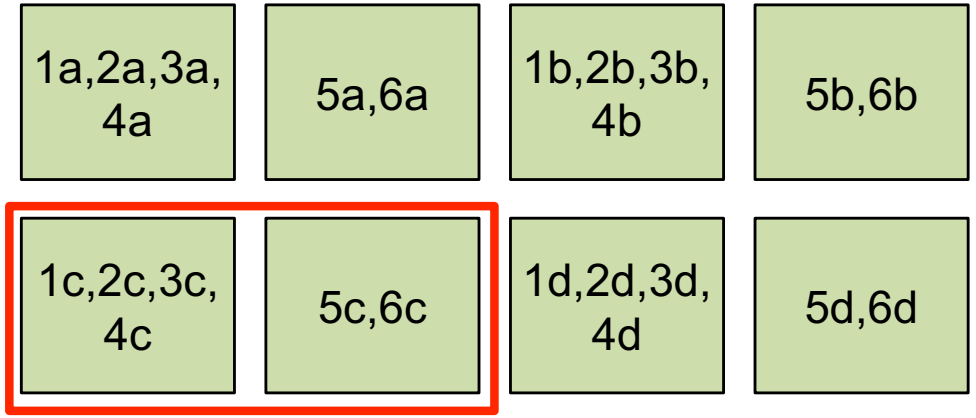
Scaling to Disk: Non-dedup. Project

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:



- 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.); popular in online analytical processing ("OLAP")
  - Rationale for col-store RDBMS (e.g., Vertica) and Parquet

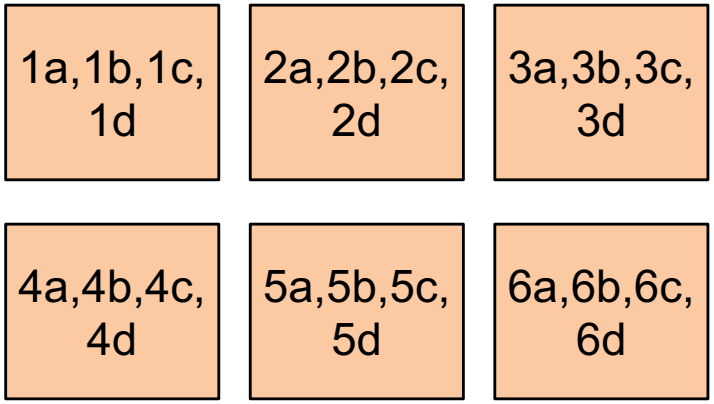
Scaling to Disk: Simple Aggregates

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:



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

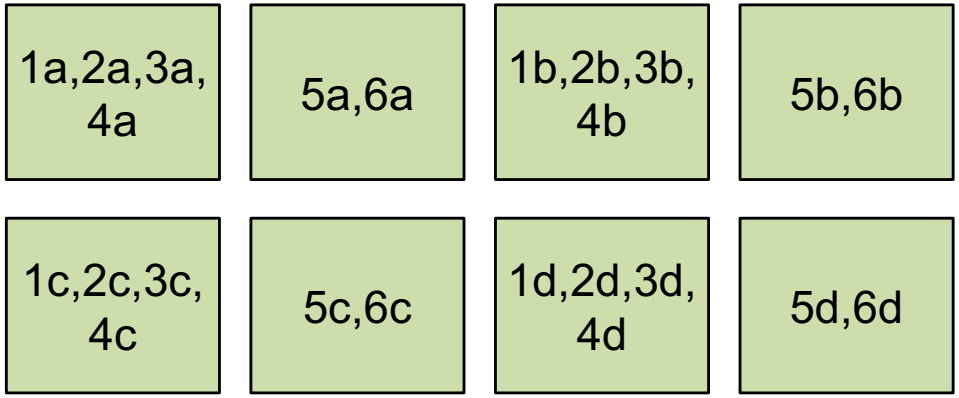
Scaling to Disk: Simple Aggregates

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:



- 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 aggregation function to each
- Typically done with a **hash table** maintained in DRAM
  - Has 1 record per group and maintains "running information" for that group's aggregation function
  - 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

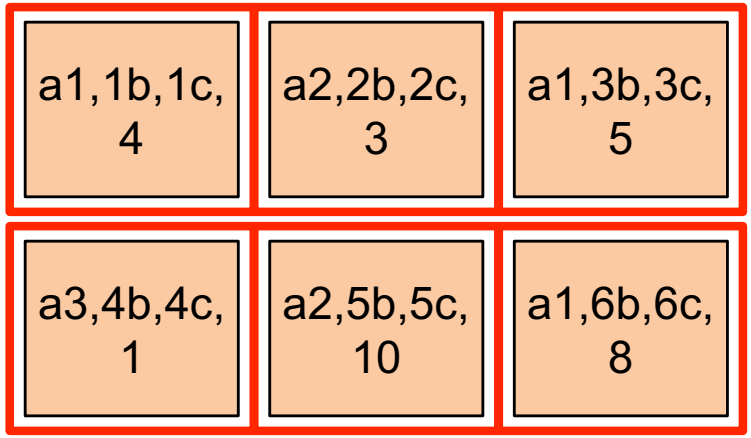
Scaling to Disk: Group By Aggregate

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:



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 might crash depending on backend implementation. 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 Data Science Operations

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

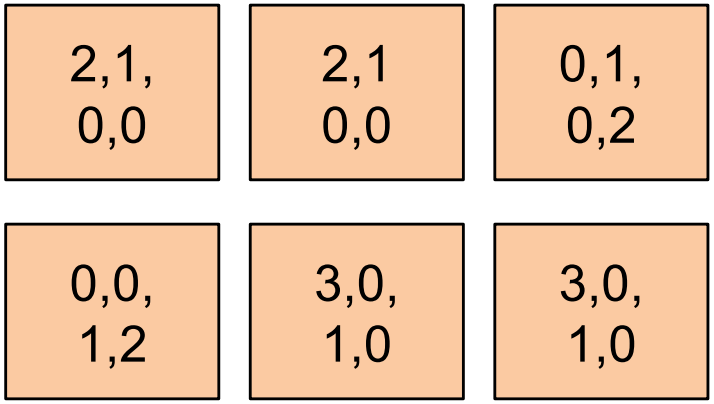
Scaling to Disk: Matrix Sum of Squares

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}

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

Row-store:



- Again, straightforward **filescan** data access pattern
  - Very similar to relational simple aggregate
  - Running info. in DRAM for sum of squares of cells
    - 0 -> 5 -> 10 -> 15 -> 20 -> 30 -> 40
- I/O cost: 6 (read) + output # pages (write)

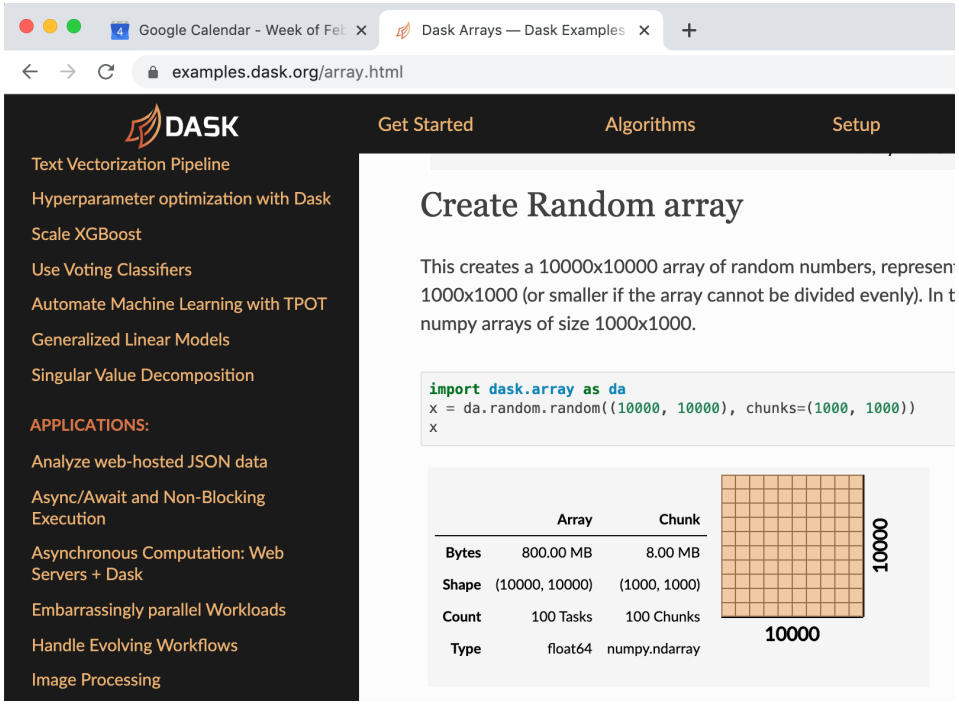
Scalable Matrix/Tensor Algebra

- In general, tiled partitioning is more common for matrix/tensor ops

- DRAM-to-disk scaling:
  - pBDR, SystemDS, and Dask Arrays for matrices
  - SciDB, Xarray for n-d arrays
- CUDA for DRAM-GPU caches scaling of matrix/tensor ops

pbdR
Programming with Big Data in R

Apache SystemDS™

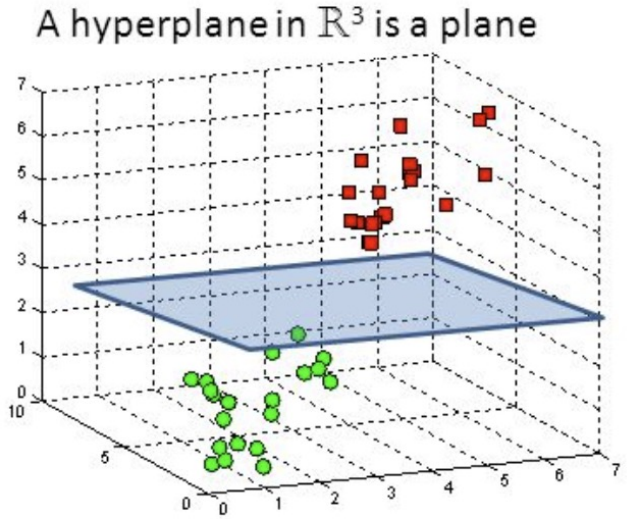


Numerical Optimization in ML

- Many regression and classification models in ML are formulated as a (constrained) *minimization* problem
  - E.g., logistic and linear regression, linear SVM, etc.
  - Aka “Empirical Risk Minimization” (ERM) approach
  - Computes “loss” of predictions over labeled examples

$$\mathbf{w}^* = argmin_{\mathbf{w}} \sum_{i=1}^{n} l(y_i, f(\mathbf{w}, x_i))$$

- Hyperplane-based models aka Generalized Linear Models (GLMs) use f() that is a scalar function of distances: $\mathbf{w}^T x_i$



Batch Gradient Descent for ML

$$L(\mathbf{w}) = \sum_{i=1}^{n} l(y_i, f(\mathbf{w}, x_i))$$

- In many cases, **loss L**() is **convex**;
- Closed-form minimization typically infeasible
- **Batch Gradient Descent:**
  - Iterative numerical procedure to find an optimal **w**
  - Initialize **w** to some value $\mathbf{w}^{(0)}$
  - Compute **gradient**:

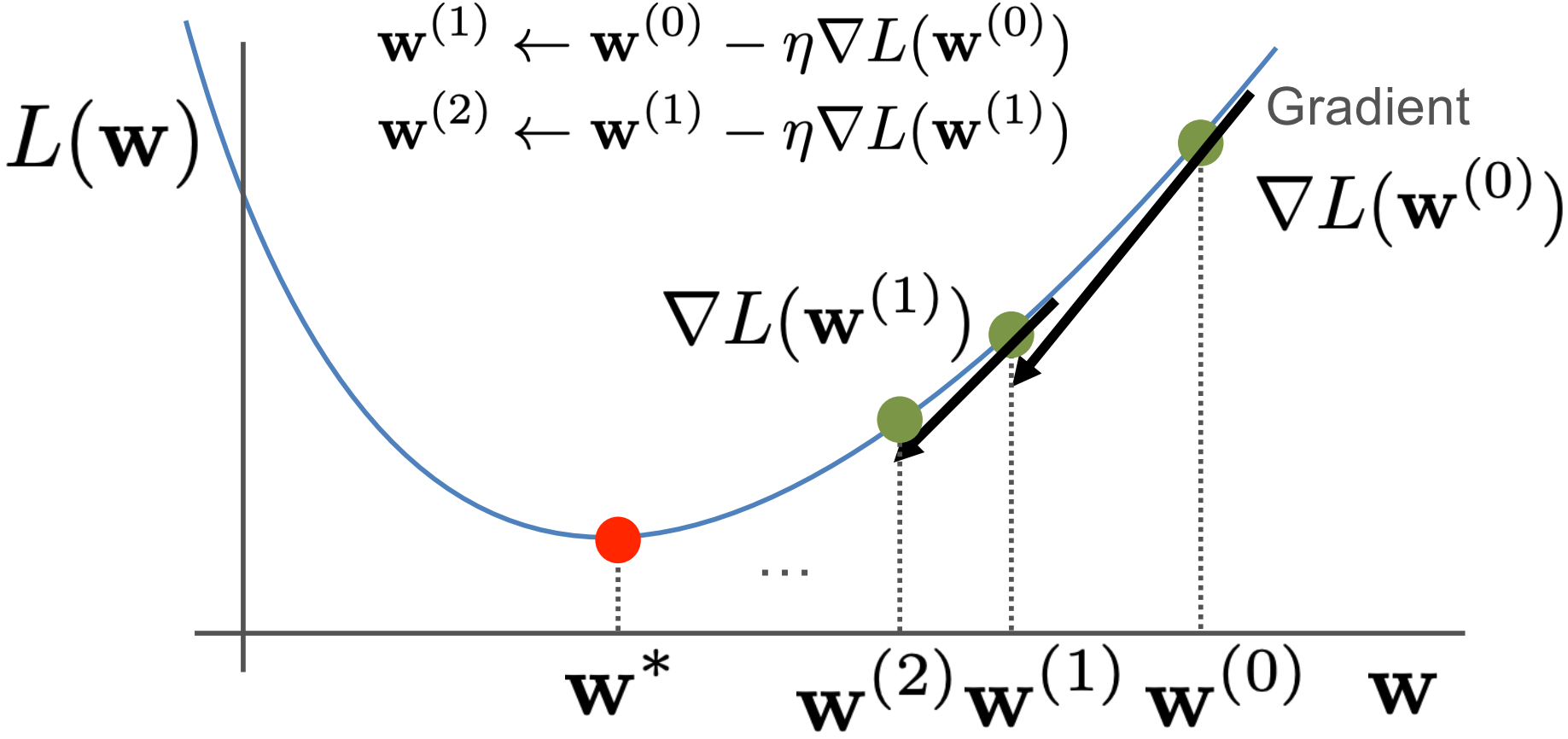
$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

  - Descend along gradient:
    (Aka **Update Rule**)

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla L(\mathbf{w}^{(k)})$$

  - Repeat until we get close to **w***, aka **convergence**

Batch Gradient Descent for ML



- Learning rate is a **hyper-parameter** selected by user or "AutoML" tuning procedures
- Number of epochs (iterations) of BGD also hyper-parameter

Data Access Pattern of BGD at Scale

- The data-intensive computation in BGD is the gradient
  - In scalable ML, dataset D may not fit in DRAM
  - Model **w** is typically (but not always) small and DRAM-resident

$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

Q: *What SQL operation is this reminiscent of?*

- Gradient is like SQL SUM over vectors (one per example)
- At each epoch, 1 **filescan** over D to get gradient
- Update of **w** happens normally in DRAM
- Monitoring across epochs for convergence needed
- Loss function L() is also just a SUM in a similar manner

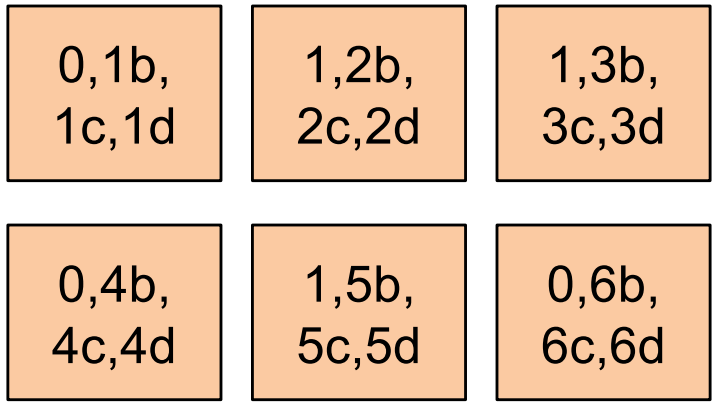
I/O Cost of Scalable BGD

$$\nabla L(\mathbf{w}^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(\mathbf{w}^{(k)}, x_i))$$

D

Y	X1	X2	X3
0	1b	1c	1d
1	2b	2c	2d
1	3b	3c	3d
0	4b	4c	4d
1	5b	5c	5d
0	6b	6c	6d

Row-store:



- Straightforward **filescan** data access pattern for SUM
  - Similar I/O behavior as non-dedup. project and simple SQL aggregates
- I/O cost: 6 (read) + output # pages (write for final **w**)

Stochastic Gradient Descent for ML

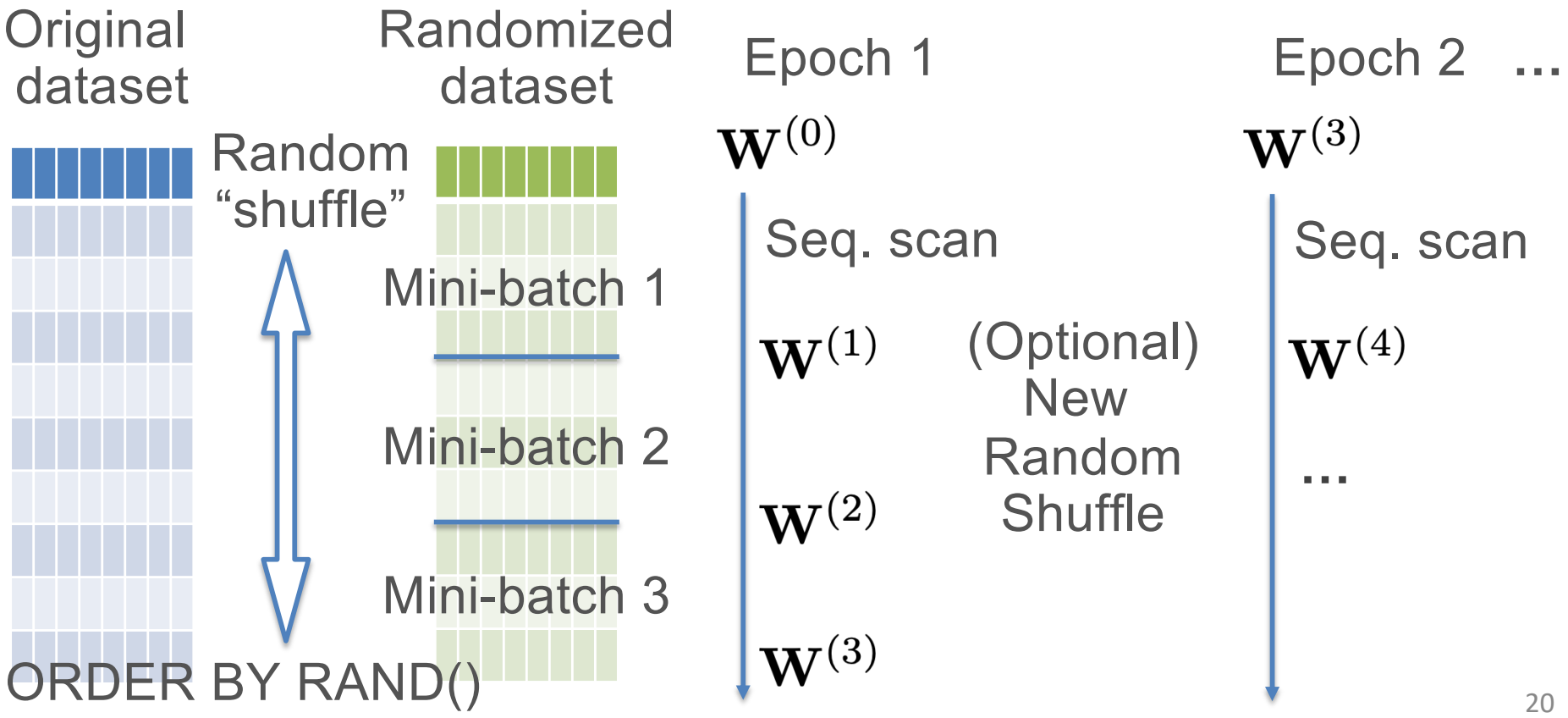
- Two key cons of BGD:
  - Often, too many epochs to reach optimal
  - Each update of **w** needs full scan: costly I/Os, full design matrix in memory

- Stochastic GD (SGD) mitigates both cons
- **Basic Idea:** Use a *sample* (**mini-batch**) of D to approximate gradient instead of "full batch" gradient
  - Done *without replacement*
  - Randomly reorder/shuffle D before every epoch
  - Sequential pass: sequence of mini-batches
- Another big pro of SGD: works better for *non-convex* loss too, especially DL
- SGD often called the "workhorse" of modern ML/DL

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



I/O Cost of (Very) Scalable SGD

- I/O cost of random shuffle is non-trivial; need so-called "external merge sort" (skipped in this course)
  - Typically amounts to 1 or 2 passes over file

- Mini-batch gradient computations: 1 **filescan** per epoch:
  - As filescan proceeds, count # examples seen, accumulate per-example gradient for mini-batch
  - Typical mini-batch sizes: 10s to 1000s
  - Orders of magnitude more model updates than BGD!
- Total I/O cost per epoch: 1 shuffle cost + 1 filescan cost
  - Often, shuffling only once upfront suffices
- Loss function L() computation is same as before (for BGD)

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

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. Which is the best layout format for 2-D structured data?
4. Briefly explain 1 pro and 1 con of SGD vs BGD.
5. Suppose you use scalable SGD to train a DL model. The dataset has 100 mil examples. You use a mini-batch size of 50. How many iterations (number of model update steps) will SGD finish in 20 epochs?
6. What is the runtime tradeoff involved in shuffle-once-upfront vs shuffle-every-epoch for SGD? Assume a physical shuffle is necessary, not an index-based shuffle.