DSC 102
Systems for Scalable Analytics

Arun Kumar

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

Access Speed
- Main Memory: ~100GB/s
- Cache: ~10GB/s
- Flash Storage: ~10GB/s
- Magnetic Hard Disk Drive (HDD): ~200MB/s

Cycles
- Access: ~10^7 - 10^8
- Main Memory: 105 - 10^6
- Cache: 100s
- Flash Storage: ~10^7 - 10^8
- HDD: ~10^7 - 10^8

Capacity
- Main Memory: ~100s
- Cache: ~10^6
- Flash Storage: ~10^7
- HDD: ~10^8

Price
- Access: ~$2/MB
- Main Memory: ~$5/GB
- Cache: ~$200/TB
- Flash Storage: ~$40/TB
- HDD: ~$10/GB
Memory Hierarchy in Action

Rough sequence of events when program is executed

Arithmetic done within CPU

Store; Retrieve

Q: What if this does not fit in DRAM?

CPU

CU

ALU

Registers

‘21’

Caches

‘21’

Commands interpreted

Bus

I/O for Display

Monitor

I/O for code

Disk

tmp.py
tmp.csv

I/O for data
Scale of Datasets in Practice

KDnuggets 2020 Poll: Largest Dataset Analyzed

N = 562
62% industry

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 \(\Rightarrow\) \(\frac{40\text{GB}}{800\text{MBps}} = 50\text{s}\)

Network with speed: 200 MB/s \(\Rightarrow\) \(\frac{40\text{GB}}{200\text{MBps}} = 200\text{s}\)
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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>2</td>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>3</td>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>4</td>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>5</td>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>6</td>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

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

Say, a page can fit only 4 cell values

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

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

Say, a page can fit only 4 cell values

Hybrid stores with 2x2 tiled layout:

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)

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: Relational Select

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

$R = \sigma_B = \left(3b\right)(R)$

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 **filenscan** 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 to Disk: Non-dedup. Project

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

**Row-store:**

- 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 **filesca**n 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

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>2a</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>3a</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>4a</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>5a</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>6a</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

Row-store:

Row-store:

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

SELECT MAX(A) FROM R

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1b</td>
<td>1c</td>
<td>4</td>
</tr>
<tr>
<td>a2</td>
<td>2b</td>
<td>2c</td>
<td>3</td>
</tr>
<tr>
<td>a1</td>
<td>3b</td>
<td>3c</td>
<td>5</td>
</tr>
<tr>
<td>a3</td>
<td>4b</td>
<td>4c</td>
<td>1</td>
</tr>
<tr>
<td>a2</td>
<td>5b</td>
<td>5c</td>
<td>10</td>
</tr>
<tr>
<td>a1</td>
<td>6b</td>
<td>6c</td>
<td>8</td>
</tr>
</tbody>
</table>

R

```sql
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)

<table>
<thead>
<tr>
<th>A</th>
<th>Running Info.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>17</td>
</tr>
<tr>
<td>a2</td>
<td>13</td>
</tr>
<tr>
<td>a3</td>
<td>1</td>
</tr>
</tbody>
</table>
Scaling to Disk: Group By Aggregate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>1b</td>
<td>1c</td>
<td>4</td>
</tr>
<tr>
<td>a2</td>
<td>2b</td>
<td>2c</td>
<td>3</td>
</tr>
<tr>
<td>a1</td>
<td>3b</td>
<td>3c</td>
<td>5</td>
</tr>
<tr>
<td>a3</td>
<td>4b</td>
<td>4c</td>
<td>1</td>
</tr>
<tr>
<td>a2</td>
<td>5b</td>
<td>5c</td>
<td>10</td>
</tr>
<tr>
<td>a1</td>
<td>6b</td>
<td>6c</td>
<td>8</td>
</tr>
</tbody>
</table>

R

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

Hash table in DRAM

<table>
<thead>
<tr>
<th>A</th>
<th>Running Info.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>4 -&gt; 9 -&gt; 17</td>
</tr>
<tr>
<td>a2</td>
<td>3 -&gt; 13</td>
</tr>
<tr>
<td>a3</td>
<td>1</td>
</tr>
</tbody>
</table>

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

❖ 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 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: Matrix Sum/Norms

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$M_{6x4}$

\[ \| M \|_2^2 \]

Row-store:

- Again, straightforward **files**can 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)
- Col-store and tiled-store also have I/O cost 6; why?
Scalable Matrix/Tensor Algebra

- In general, tiled partitioning is more common for matrix/tensor ops
  - More DRAM/cache-efficient implementations

- 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
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

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

- Hyperplane-based models aka Generalized Linear Models (GLMs) use $f()$ that is a scalar function of distances: $\mathbf{w}^T \mathbf{x}_i$
Batch Gradient Descent for ML

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

- In many cases, loss function \( l() \) is convex; so is \( L() \)
- Closed-form minimization typically infeasible
- **Batch Gradient Descent:**
  - Iterative numerical procedure to find an optimal \( w \)
  - Initialize \( w \) to some value \( w^{(0)} \)
  - Compute **gradient:**
    \[ \nabla L(w^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(w^{(k)}, x_i)) \]
  - Descend along gradient:
    (Aka **Update Rule**)
    \[ w^{(k+1)} \leftarrow w^{(k)} - \eta \nabla L(w^{(k)}) \]
  - Repeat until we get close to \( w^* \), aka **convergence**
Batch Gradient Descent for ML

\[ L(w) \]

\[ \mathbf{w}^{(1)} \leftarrow \mathbf{w}^{(0)} - \eta \nabla L(\mathbf{w}^{(0)}) \]
\[ \mathbf{w}^{(2)} \leftarrow \mathbf{w}^{(1)} - \eta \nabla L(\mathbf{w}^{(1)}) \]

- Learning rate is a **hyper-parameter** selected by user or “AutoML” tuning procedures
- Number of iterations/epochs of BGD also hyper-parameter
The data-intensive computation in BGD is the gradient.

In scalable ML, dataset D may not fit in DRAM.

Model $w$ is typically small and DRAM-resident.

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.

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

Q: What SQL op is this reminiscent of?
I/O Cost of Scalable BGD

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

<table>
<thead>
<tr>
<th>Y</th>
<th>X1</th>
<th>X2</th>
<th>X3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1b</td>
<td>1c</td>
<td>1d</td>
</tr>
<tr>
<td>1</td>
<td>2b</td>
<td>2c</td>
<td>2d</td>
</tr>
<tr>
<td>1</td>
<td>3b</td>
<td>3c</td>
<td>3d</td>
</tr>
<tr>
<td>0</td>
<td>4b</td>
<td>4c</td>
<td>4d</td>
</tr>
<tr>
<td>1</td>
<td>5b</td>
<td>5c</td>
<td>5d</td>
</tr>
<tr>
<td>0</td>
<td>6b</td>
<td>6c</td>
<td>6d</td>
</tr>
</tbody>
</table>

- 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 \( w \))

Row-store:
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
- 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 well for *non-convex* loss too, especially DL; BGD does not
- SGD often called the “workhorse” of modern ML/DL
Access Pattern of Scalable SGD

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

Sample mini-batch from dataset without replacement
I/O Cost of 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)
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
Peer Instruction Activity

(Switch slides)
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?

6. Briefly explain 1 pro and 1 con of SGD vs. BGD.

7. Suppose you use scalable SGD to train a DL model. The dataset has 100 million examples. Mini-batch size is set to 50. How many iterations (number of model update steps) will SGD finish in 20 epochs?

8. What is the precise runtime tradeoff involved in shuffle-once-upfront vs. shuffle-every-epoch for SGD?
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
Optional: Another Example of Scaling Data Science Operations
Not included in syllabus
## Scaling to Disk: Gramian Matrix

### Matrix $M_{6x4}$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

### Gramian Matrix $M^T M$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1,</td>
<td>2,1,</td>
<td>0,1,</td>
</tr>
<tr>
<td>0,0,</td>
<td>0,0,</td>
<td>0,2,</td>
</tr>
<tr>
<td>0,0,</td>
<td>3,0,</td>
<td>3,0,</td>
</tr>
<tr>
<td>1,2,</td>
<td>1,0,</td>
<td>1,0,</td>
</tr>
</tbody>
</table>

#### Observations:

- A bit tricky, since output may not fit entirely in DRAM
- Similar to GROUP BY difficult case
- Output here is 4x4, i.e., 4 pages; only 3 can be in DRAM!
- Each row will need to update entire output matrix
- Row-store can be a poor fit for such matrix algebra
- What about col-store or tiled-store?
Scaling to Disk: Gramian Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

$M_{6\times 4}$

2x2 Tiled store

$M^T M$

- Read A, C, E one by one to get $O_1 = A^T A + C^T C + E^T E$; $O_1$ is incrementally computed; write $O_1$ out; I/O: 3 (r) + 1 (w)
- Likewise with B, D, F for $O_4$; I/O: 3 (r) + 1 (w)
- Read A, B and put $A^T B$ in $O_2$; read C, D to add $C^T D$ to $O_2$; read E, F to add $E^T F$ to $O_2$; write $O_2$ out; I/O: 6 + 1
- Likewise with B, A; D, C; F, E for $O_3$; I/O: 6 + 1
- Max I/O cost: 18 (r) + 4(w); scalable on both dimensions!