Data Infrastructure

CSCI 5942

Lecture 6

College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

Agenda

- Wrap up discussion of Roofline Model
- Today: “Offline” data infrastructure
- Next class: “Online” data infrastructure

Hardware bottlenecks

1,000 H100s × 30 days × 40% MFU?

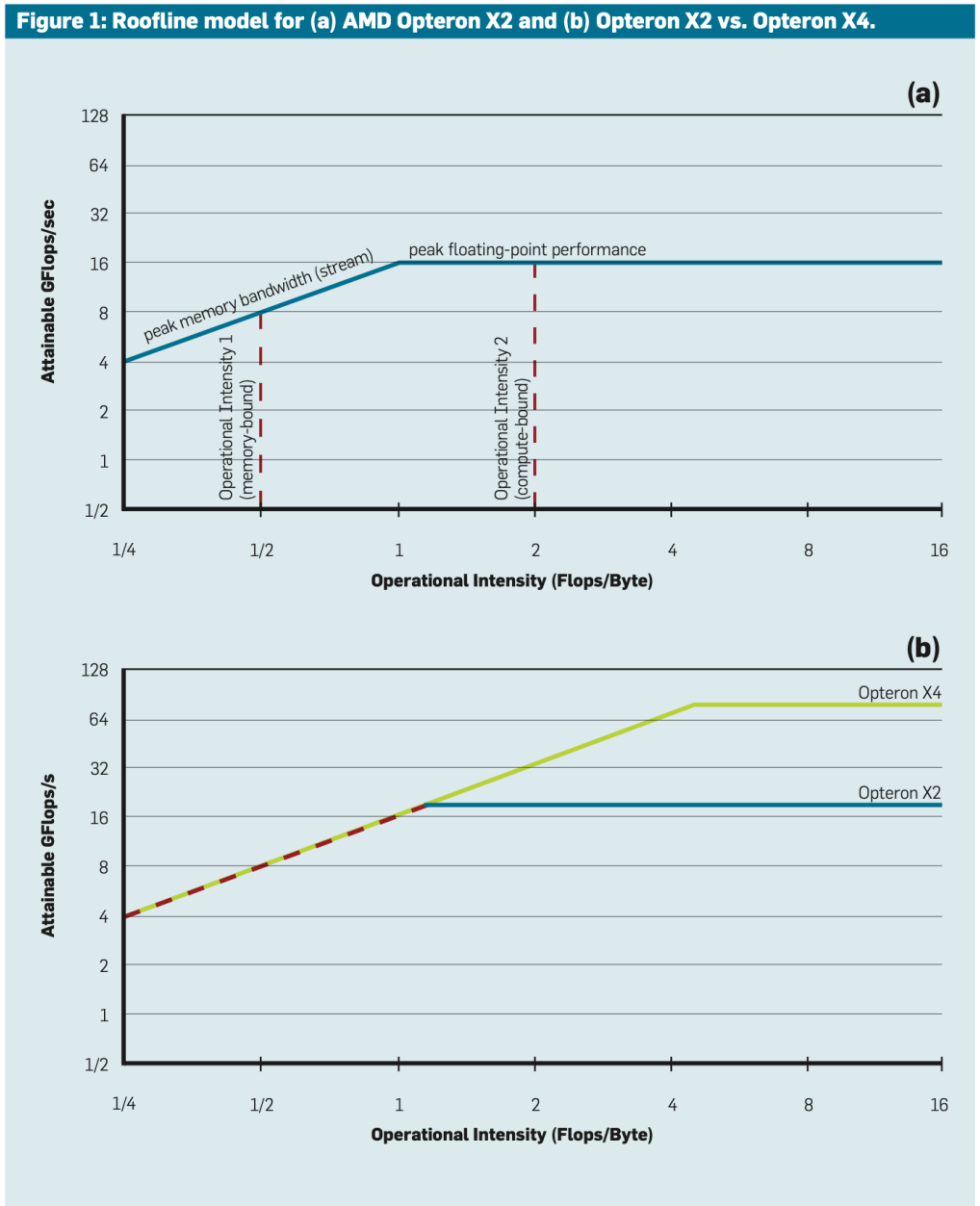
as measured; imagine why?

MFU is model FLOPs utilization: the fraction of the chips' peak rate the run actually reaches.

Hardware bottlenecks

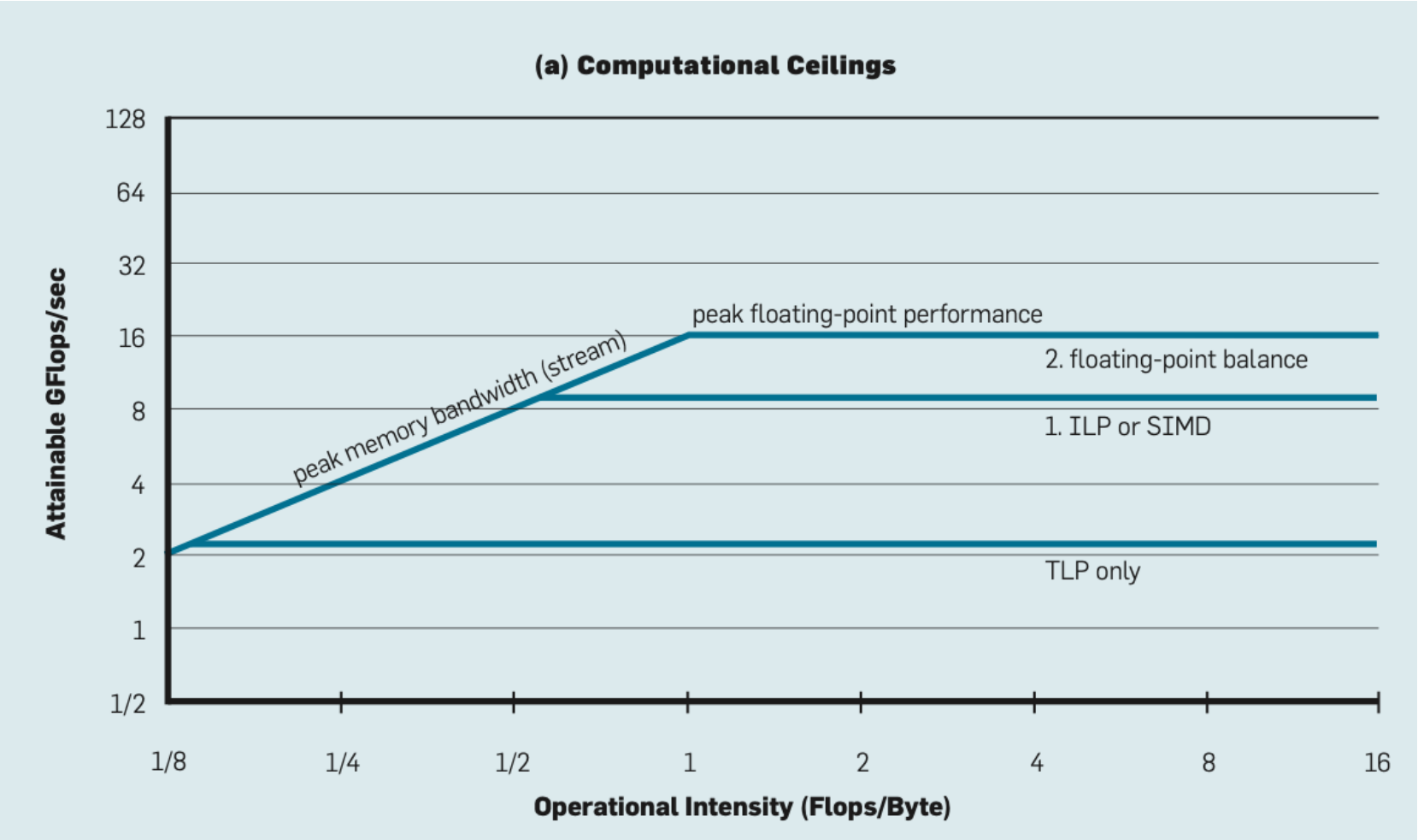
- How can we reason about if a specific model/algorithm/kernel/function is bound by off-chip memory or compute?

Roofline Model



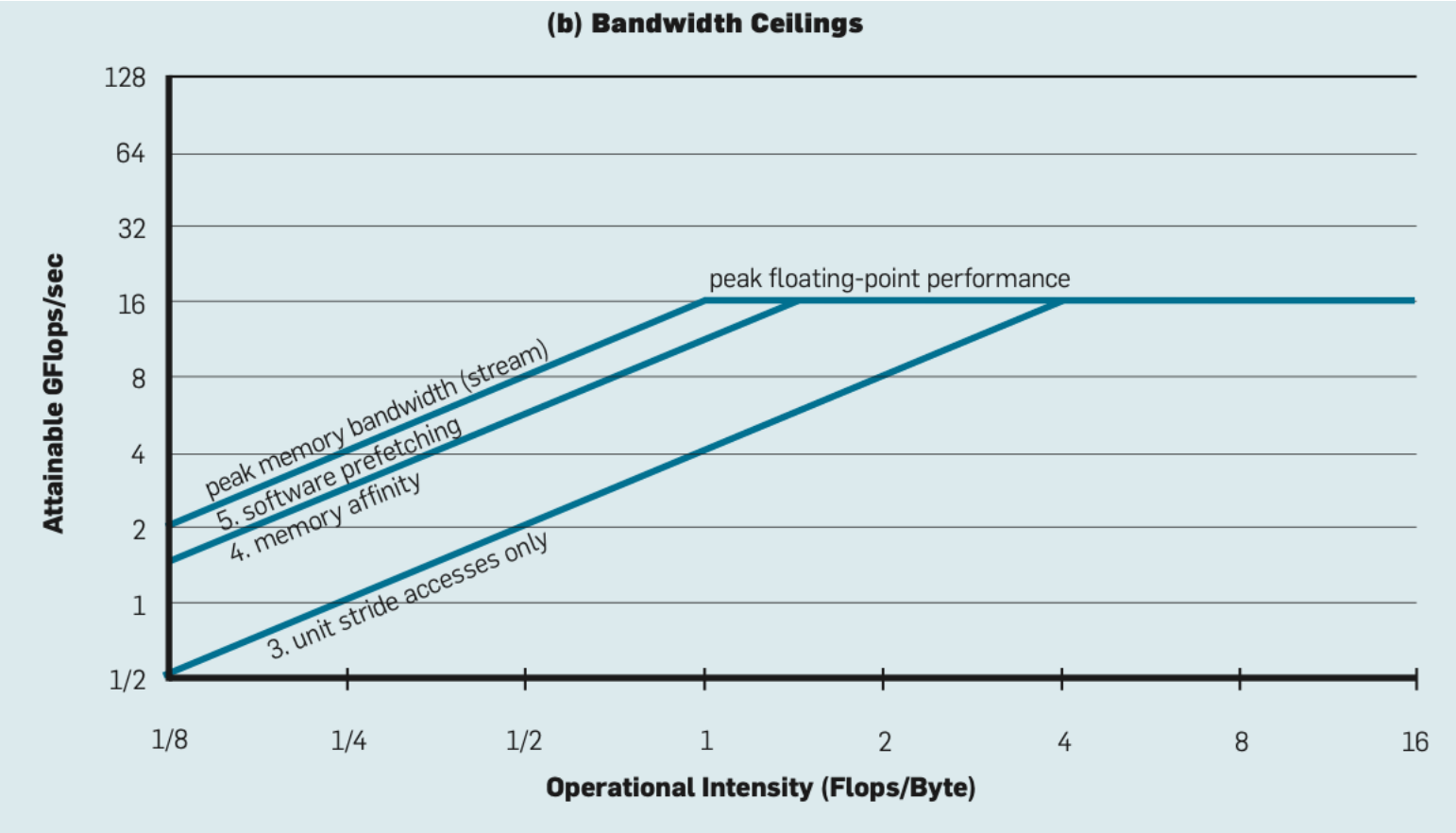
https://dl.acm.org/doi/pdf/10.1145/1498765.1498785

Roofline Model



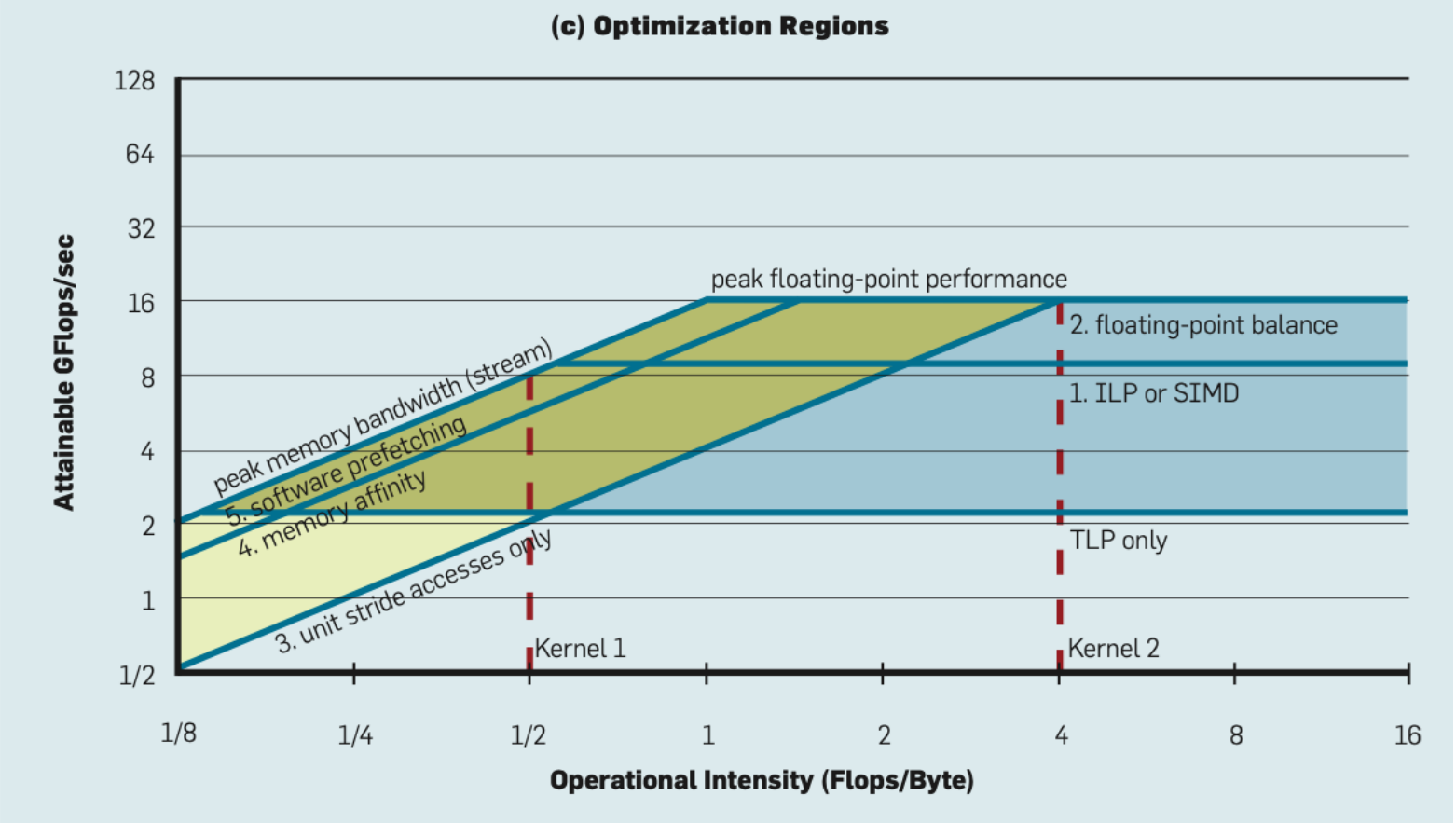
https://dl.acm.org/doi/pdf/10.1145/1498765.1498785

Roofline Model



https://dl.acm.org/doi/pdf/10.1145/1498765.1498785

Roofline Model



https://dl.acm.org/doi/pdf/10.1145/1498765.1498785

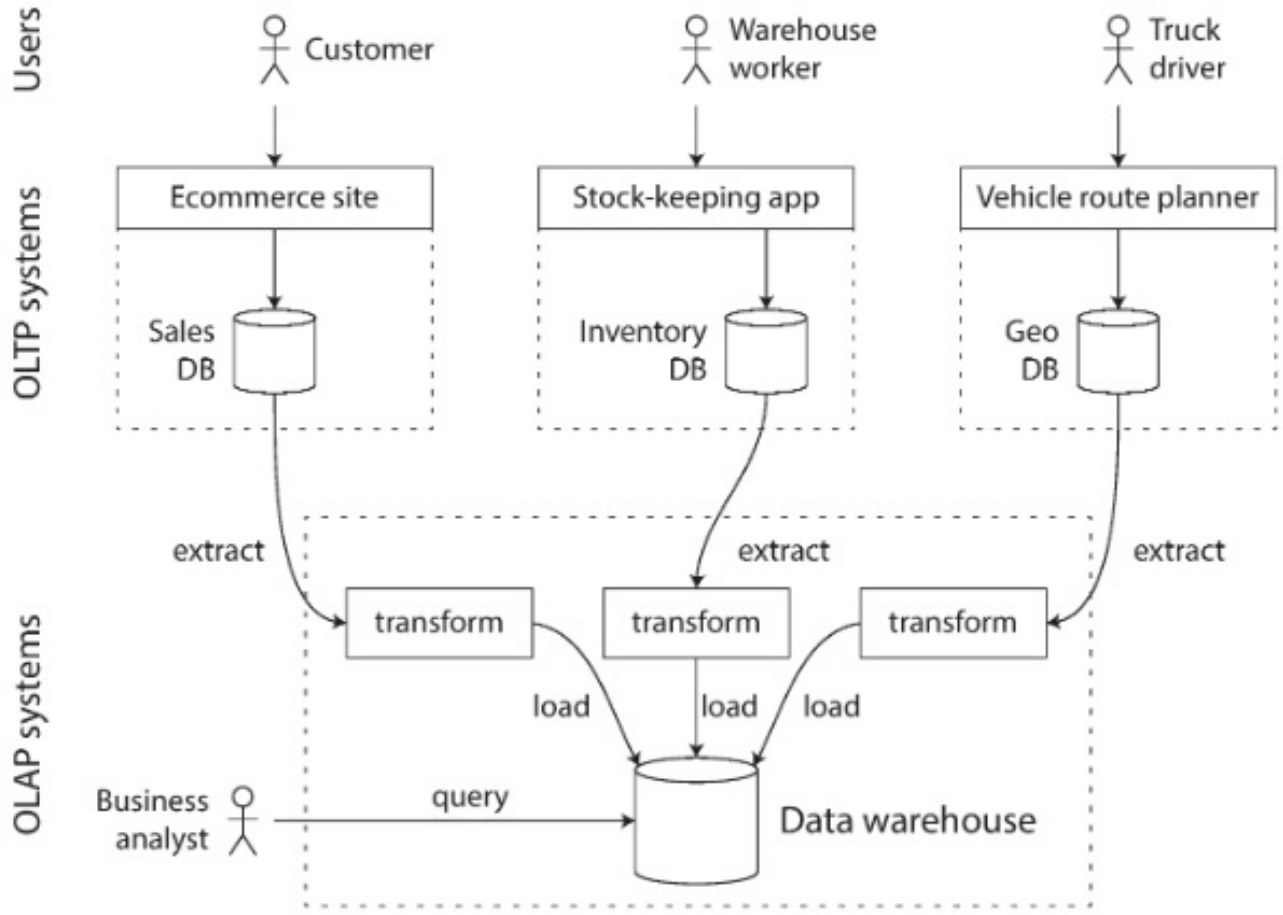
Recap: The Training Data Pipeline

Data Engineering

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

OLTP -> OLAP

Data Warehouses

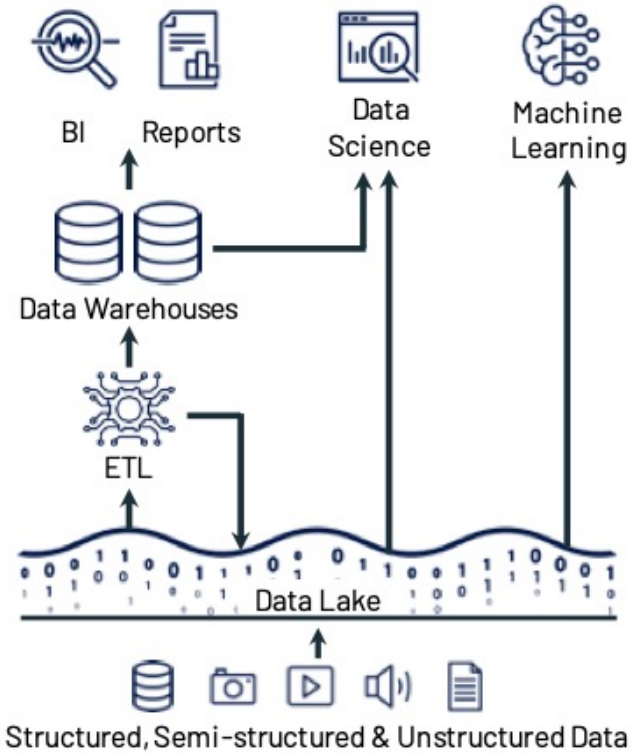


Issues with Data Warehouses

- DW focuses on structured, relational data with a schema-on-write
  - Focused on business intelligence, analytical queries
  - Expensive: Managed by some RDBMS
    - Teradata, Vertica, SAP Hana, RedShift, …
- What about ML?
  - Lots of unstructured data (videos, images, speech, …)
  - High volume: can’t afford to put everything into warehouse

Data Lake

- Land raw business data into common, low-cost storage (e.g., object store)
  - Schema-on-Read
  - ETL to data warehouse for BI
- Open file formats and systems
  - Parquet, ORC
  - HDFS, Spark, MapReduce, Hive
  - Cloud: S3, GCS, ADLS



Issues with Data Lakes

- Consistency/staleness between data lake and downstream data warehouse
- Open formats not as optimized as proprietary storage
- Paying twice for storage (lake + warehouse)
- For ML: lose data warehouse features such as indexing and versioning

Data Lakehouse

- TL;DR: Transactional metadata layer on top of data lake

Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics

Michael Armbrust[1], Ali Ghodsi[1,2], Reynold Xin[1], Matei Zaharia[1,3]
[1]Databricks, [2]UC Berkeley, [3]Stanford University

Abstract

This paper argues that the data warehouse architecture as we know it today will wither in the coming years and be replaced by a new architectural pattern, the Lakehouse, which will (i) be based on open direct-access data formats, such as Apache Parquet, (ii) have first-class support for machine learning and data science, and (iii) offer state-of-the-art performance. Lakehouses can help address several major challenges with data warehouses, including data staleness, reliability, total cost of ownership, data lock-in, and limited use-case support. We discuss how the industry is already moving toward Lakehouses and how this shift may affect work in data management. We also report results from a Lakehouse system using Parquet that is competitive with popular cloud data warehouses on TPC-DS.

1 Introduction

This paper argues that the data warehouse architecture as we know

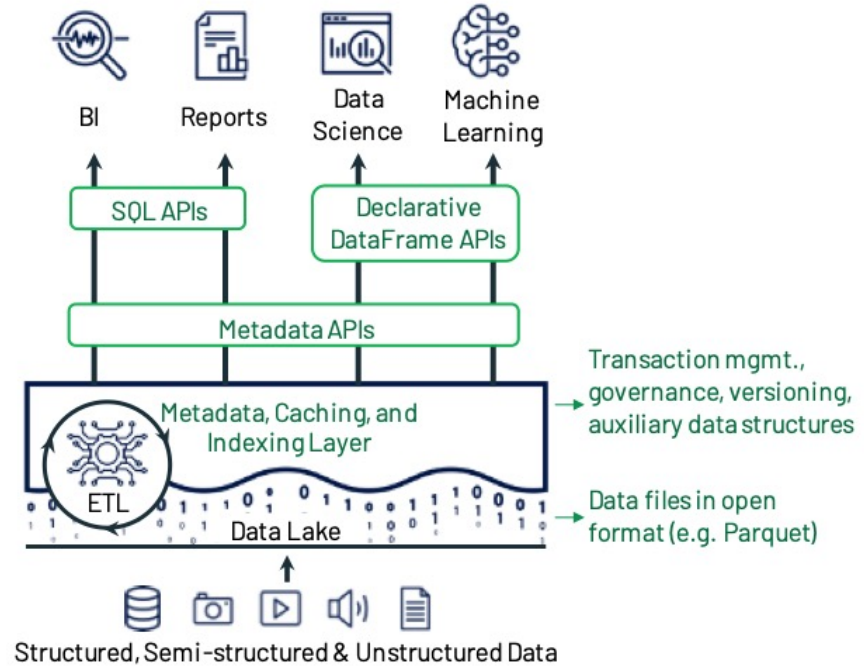
quality and governance downstream. In this architecture, a small subset of data in the lake would later be ETLed to a downstream data warehouse (such as Teradata) for the most important decision support and BI applications. The use of open formats also made data lake data directly accessible to a wide range of other analytics engines, such as machine learning systems [30, 37, 42].

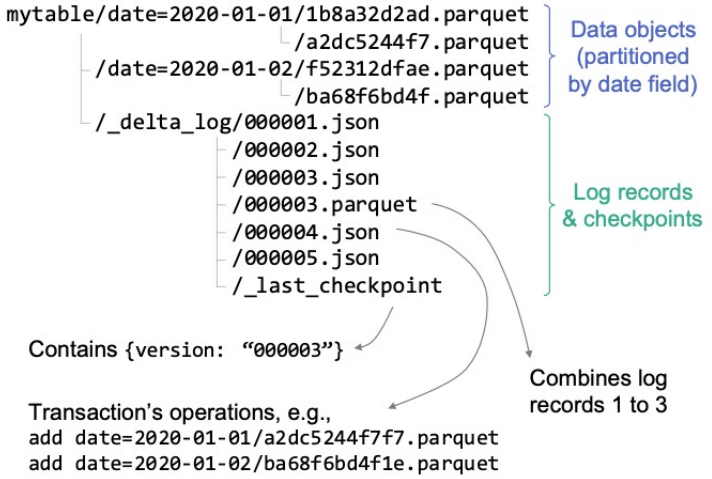
From 2015 onwards, cloud data lakes, such as S3, ADLS and GCS, started replacing HDFS. They have superior durability (often >10 nines), geo-replication, and most importantly, extremely low cost with the possibility of automatic, even cheaper, archival storage, e.g., AWS Glacier. The rest of the architecture is largely the same in the cloud as in the second generation systems, with a downstream data warehouse such as Redshift or Snowflake. This two-tier data lake + warehouse architecture is now dominant in the industry in our experience (used at virtually all Fortune 500 enterprises).

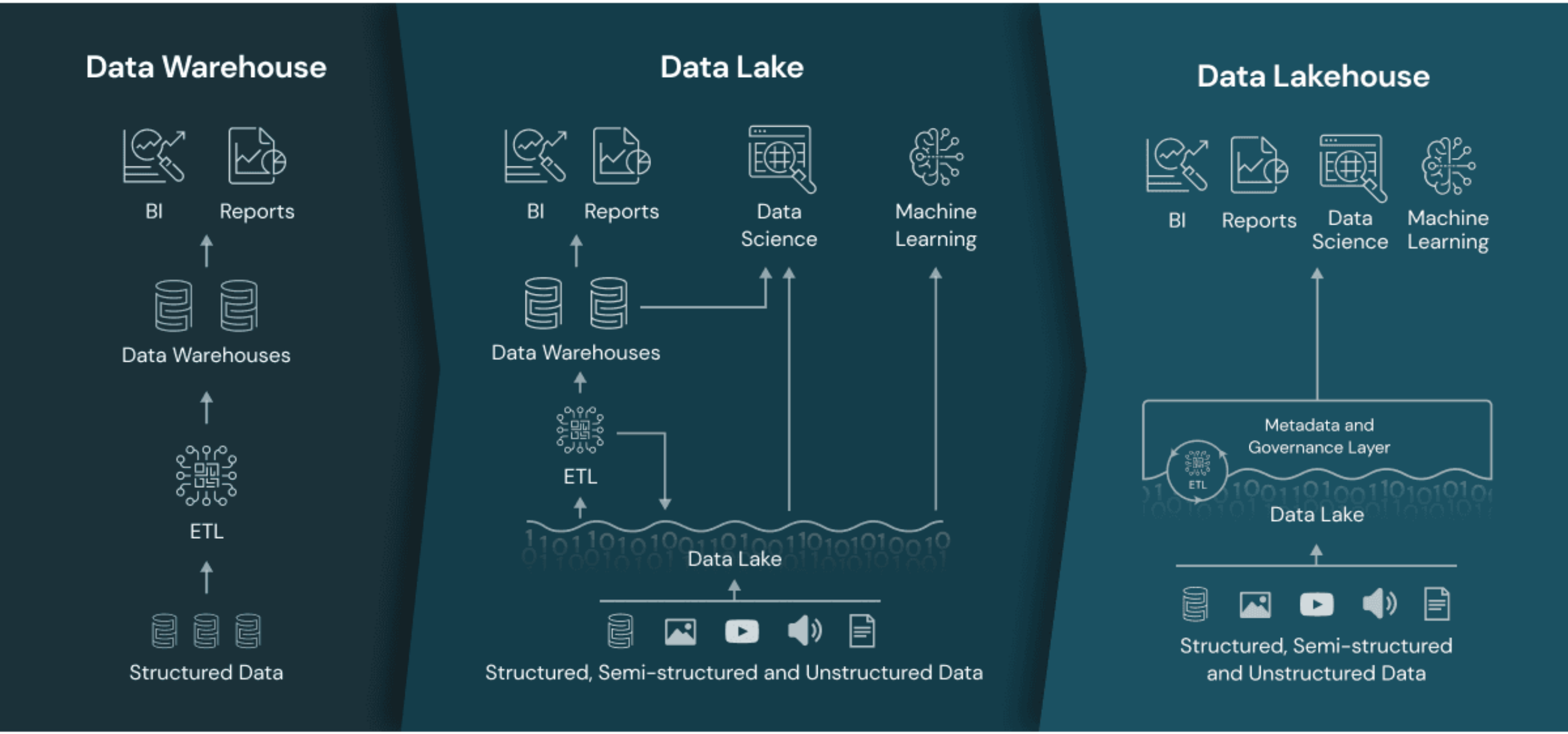
This brings us to the challenges with current data architectures. While the cloud data lake and warehouse architecture is ostensibly cheap due to separate storage (e.g., S3) and compute (e.g., Redshift), a two-tier architecture is highly complex for users. In the first gener

Data Lakehouse

- Goal: DBMS features on top of low-cost, directly-accessible storage
- *1) Metadata layer* on object store
  - See: Delta Lake, Apache Iceberg
- *2) Performance optimizations*
  - Separate storage / compute
  - Caching, auxiliary indexes, record ordering
- *3) ML Interfaces*
  - Map DataFrame API to SQL query plans
    - -> apply SQL optimizer






Data Warehouse
BI
Reports
Data Warehouses
ETL
Structured Data
Data Lake
BI
Reports
Data Science
Machine Learning
Data Warehouses
ETL
Data Lake
Structured, Semi-structured and Unstructured Data
Data Lakehouse
BI
Reports
Data Science
Machine Learning
Metadata and Governance Layer
ETL
Data Lake
Structured, Semi-structured and Unstructured Data

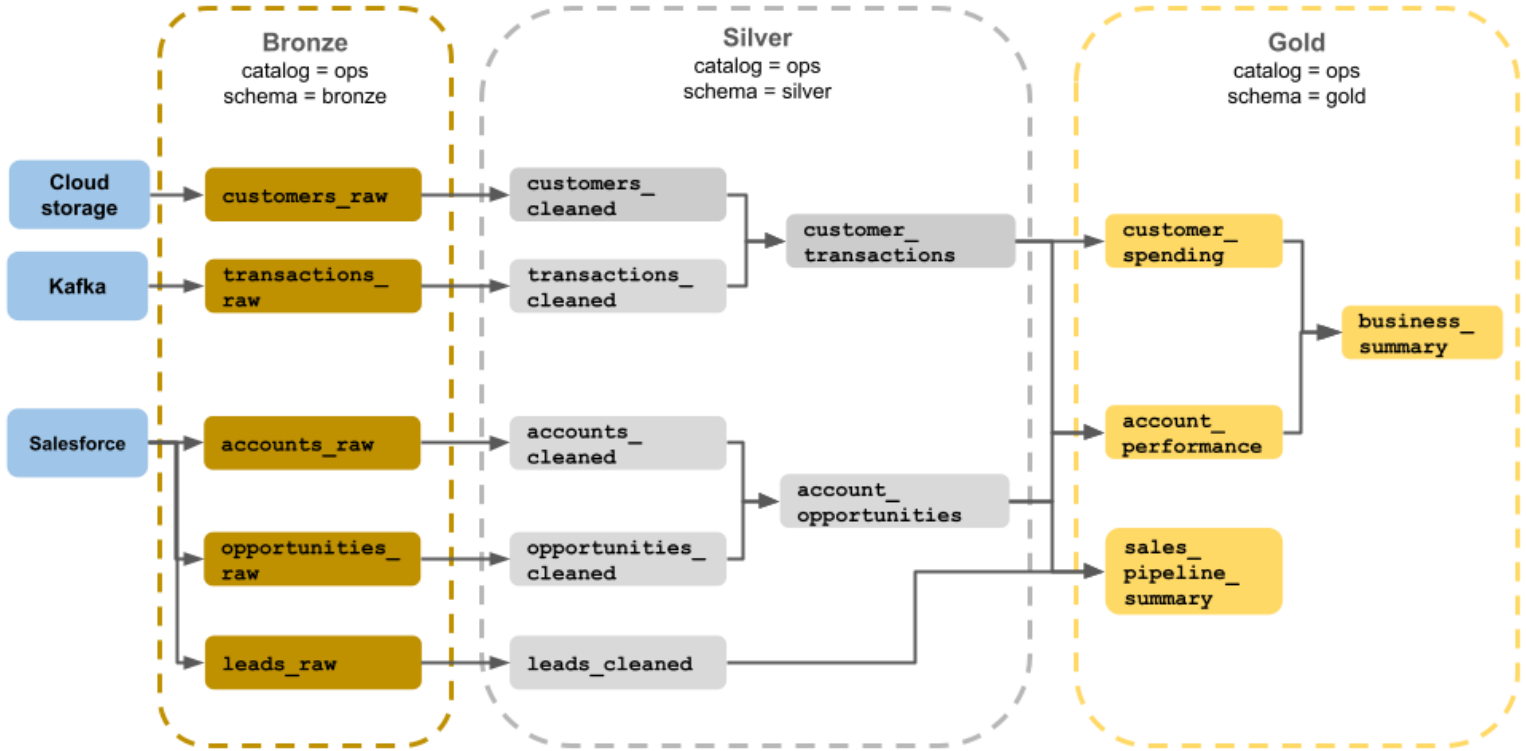
Lake, Warehouse, Lakehouse

	Data lake	Data warehouse	Lakehouse
Schema	On read	On write	On write, enforced by table format
Storage	Object store, open formats	Proprietary, coupled to engine	Object store, open formats
Transactions	None	Full ACID	ACID via metadata layer
Cost profile	Cheapest per TB	Expensive per TB and per query	Lake storage, warehouse semantics
ML fit	Good for raw + unstructured; no guarantees	Poor for unstructured; hard to version	Current default for training data

Pipeline Architecture: ETL vs ELT

	ETL — extract, transform, load	ELT — extract, load, transform
Transform	Before storage	After storage, on query
What lands	Clean, smaller, opinionated	Raw, larger, uncommitted
Schema change	Expensive — reprocess history	Cheap — schema on read
Governance	Easier: everything is structured	Harder: raw zone needs its own controls
ML use	Feature engineering with settled definitions	Exploration, and training on raw signal
Cost profile	Higher engineering cost up front	Higher storage cost, lower engineering

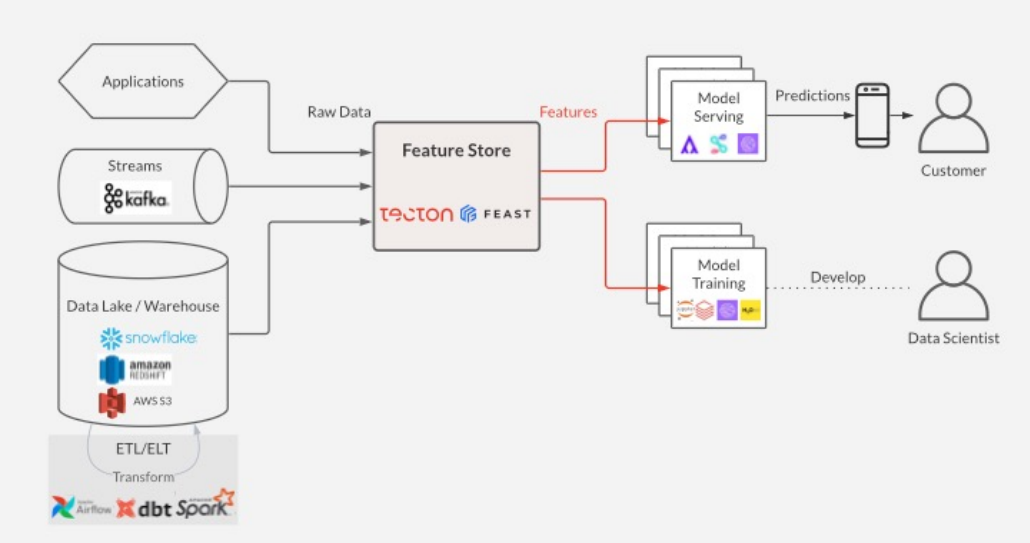
Medallion Lakehouse Architecture

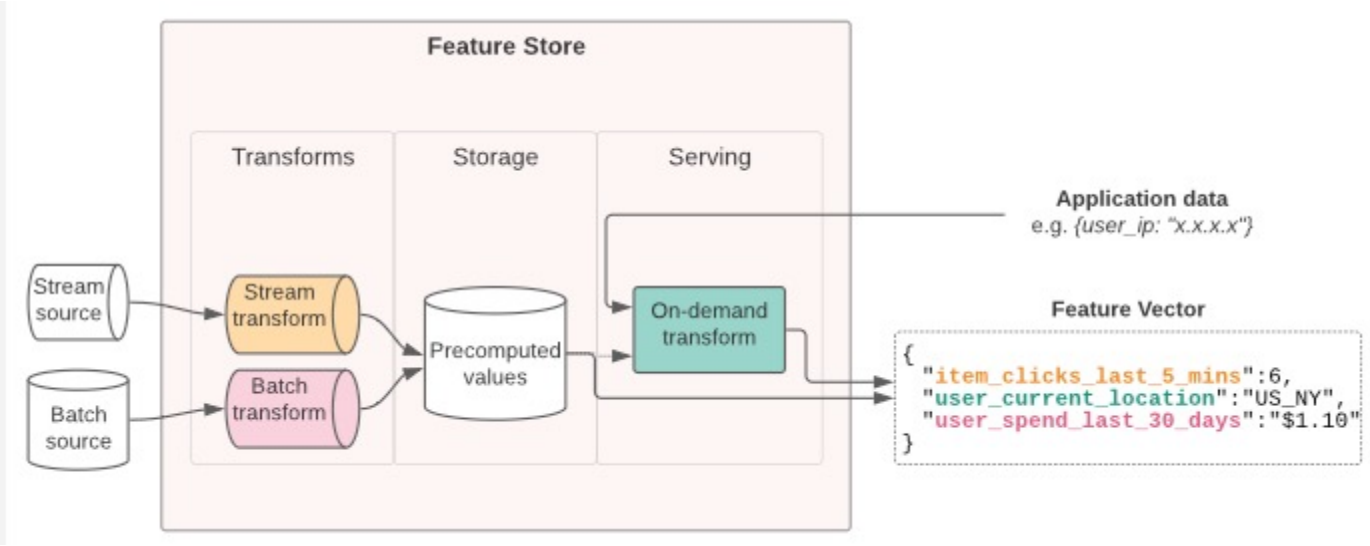


https://learn.microsoft.com/en-us/azure/databricks/lakehouse/medallion

Feature Stores

- Centralized data platform to store, manage, and serve curated input variables (features) across training and production environments.





Why might we want a feature store?

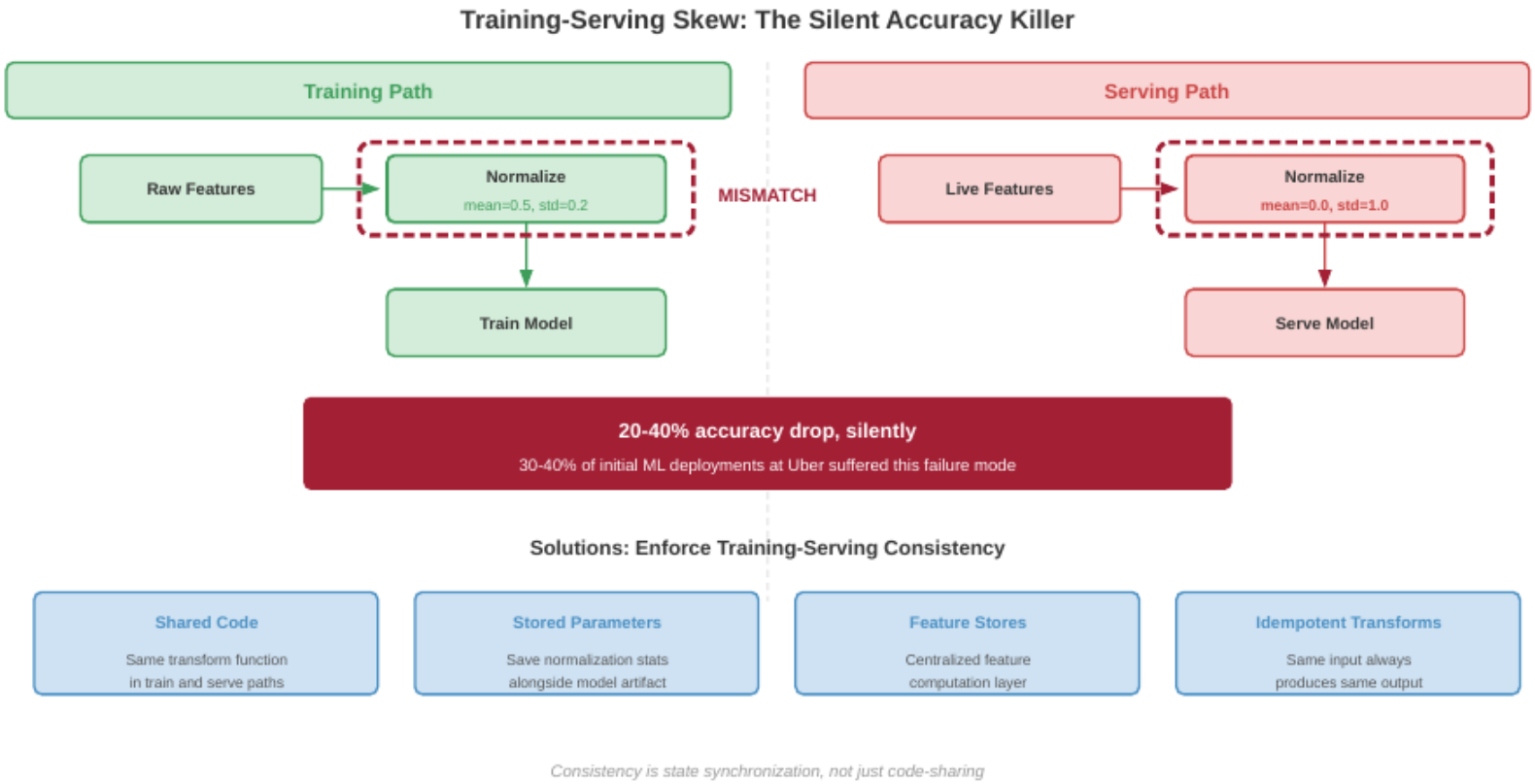
Why might we want a feature store?

- Duplicated Effort: Share/reuse feature pipelines (break “data silos”)

- Optimize performance for downstream needs
  - Training: Large “batch” reads -> high throughput
  - Serving: Low-latency, up-to-date values

Why might we want a feature store?

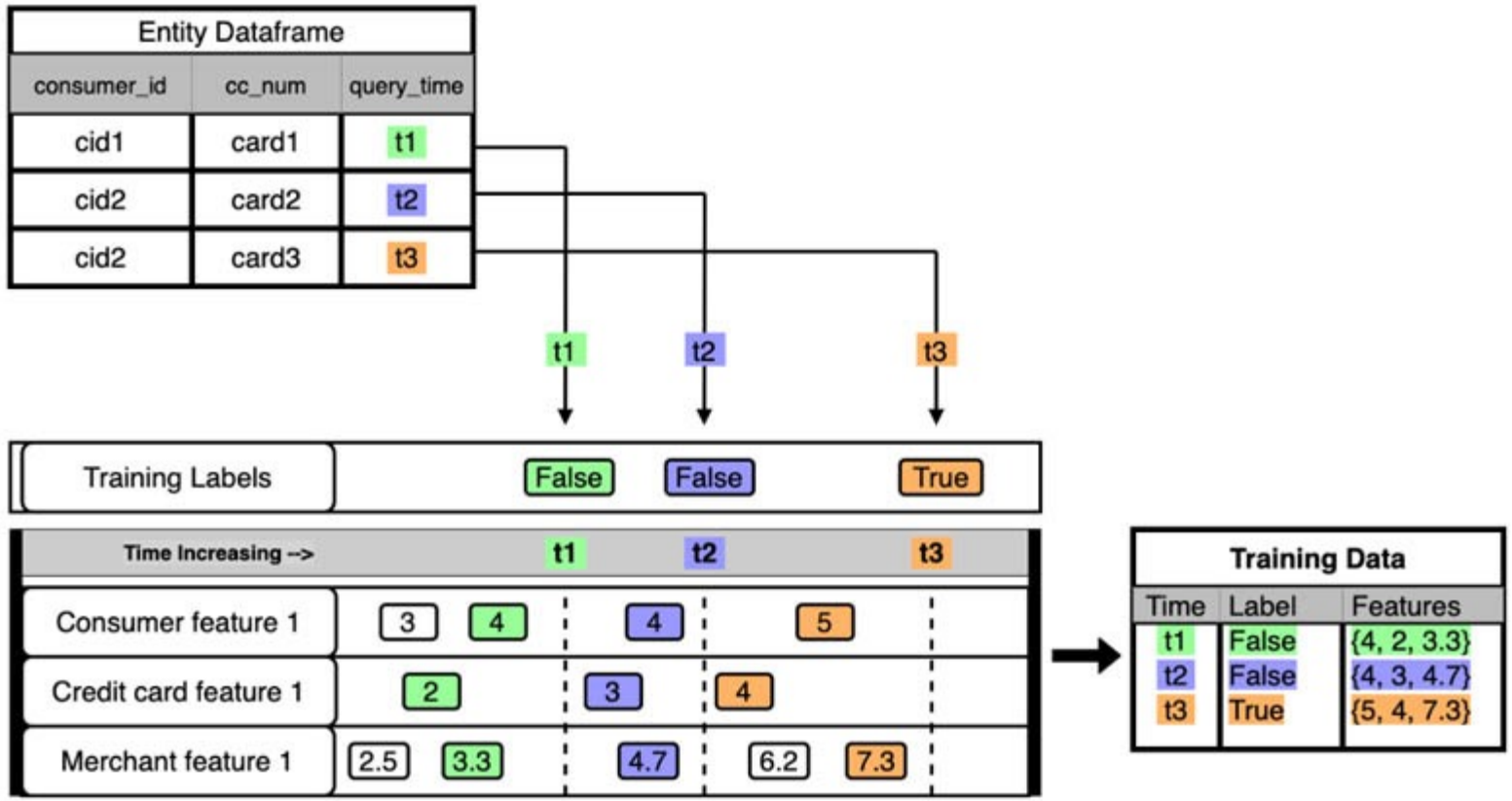
- Eliminate ”Training-Serving Skew”: Consistent transform logic across training and inference



https://mlsysbook.ai/vol1/data_engineering/data_engineering.html

Why might we want a feature store?

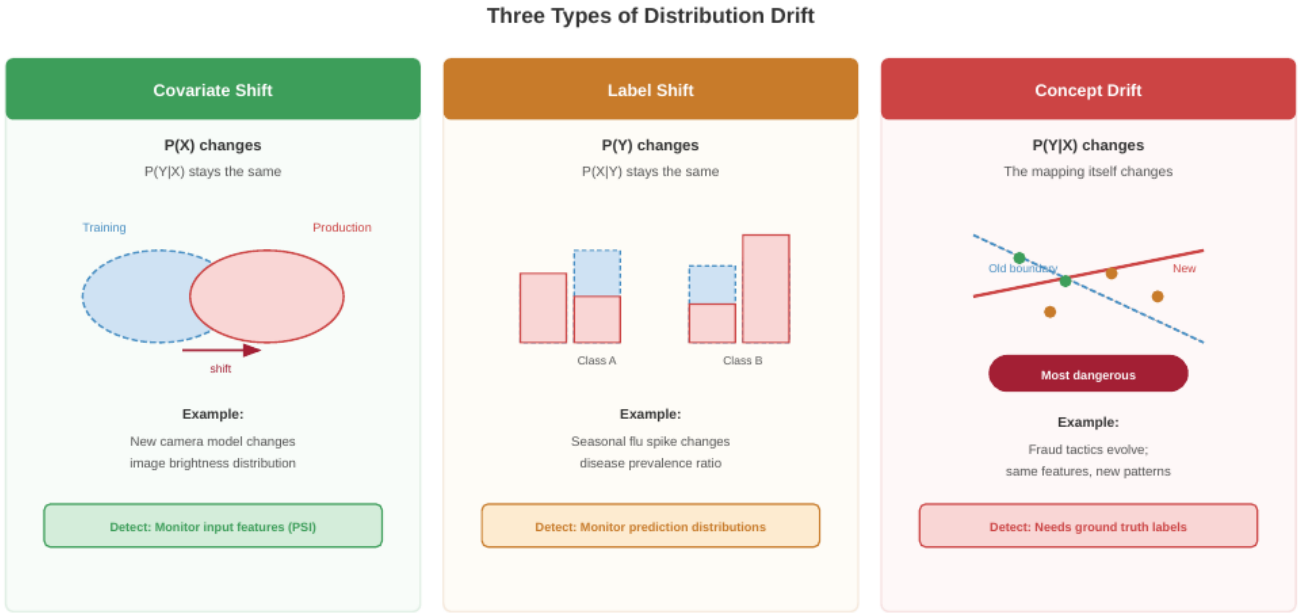
- Ensure “point-in-time” correctness
  - To train an accurate model, each training sample has to contain feature values that were present **when the label was generated**



https://aws.amazon.com/blogs/machine-learning/build-accurate-ml-training-datasets-using-point-in-time-queries-with-amazon-sagemaker-feature-store-and-apache-spark/

Why might we want a feature store?

- Centralized point to monitor for data drift



Key: Concept drift is the most dangerous — rules change but detection requires ground truth labels that arrive late.

https://mlsysbook.ai/vol1/data_engineering/data_engineering.html

Storage Models

- Suppose we wanted to store a training dataset with millions of records – e.g. (userID, feature_1, …, feature_N, label). What file should we store this dataset in?

Storage Models: N-ary Storage Model

- Row store (n-ary storage model)
  - All attributes for a single record contiguously
- Ideal for
  - Workloads which need random access to individual records
    - e.g. INSERT INTO users VALUES (A,B,C)
  - Workloads which stream the **entire** record

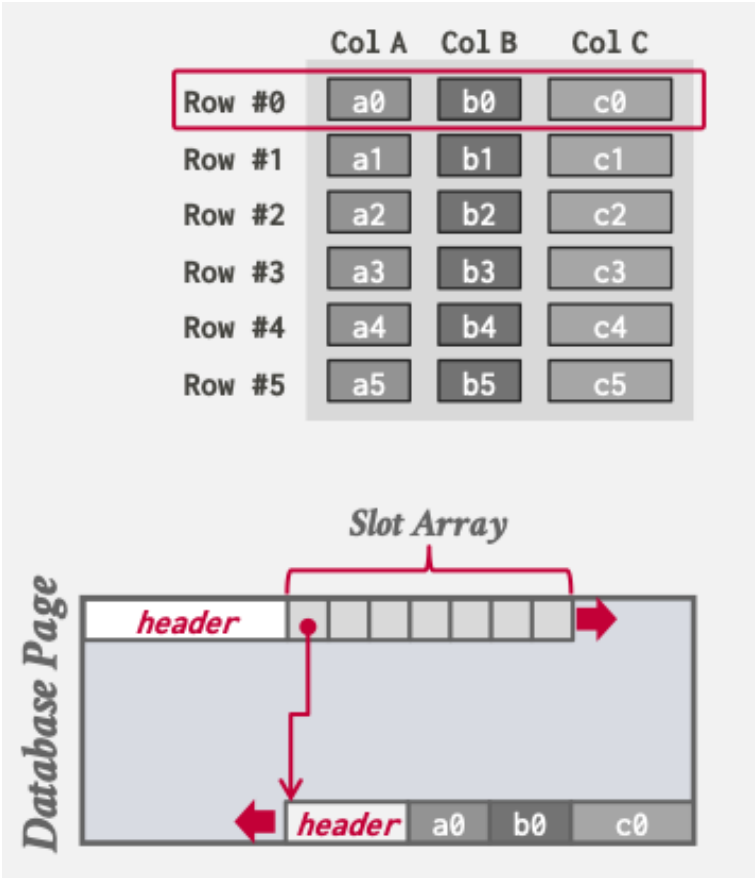


Figure from CMU 15-445/645

Storage Models (in a nutshell)

- Column store (Decomposition storage model)
  - Store single attribute for all records contiguously in block
  - Ideal for OLAP workloads which scan subset of table's attributes
    - E.g., SELECT AVG(salary) FROM employees WHERE department = 'Engineering'

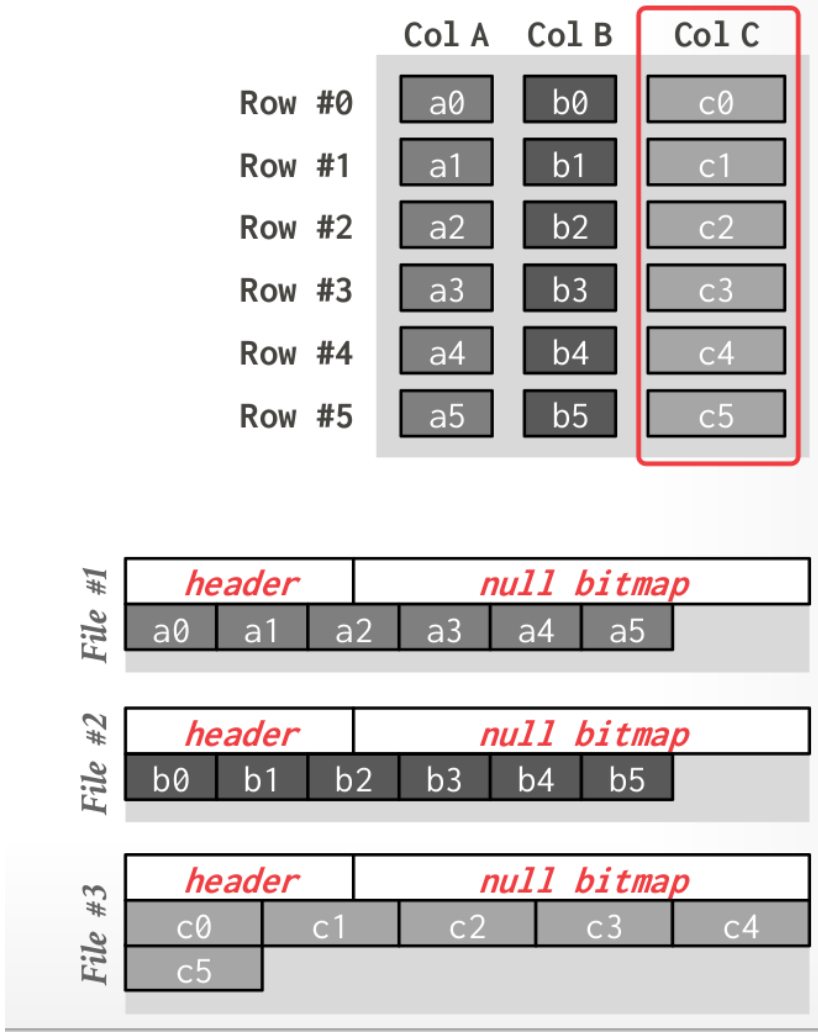
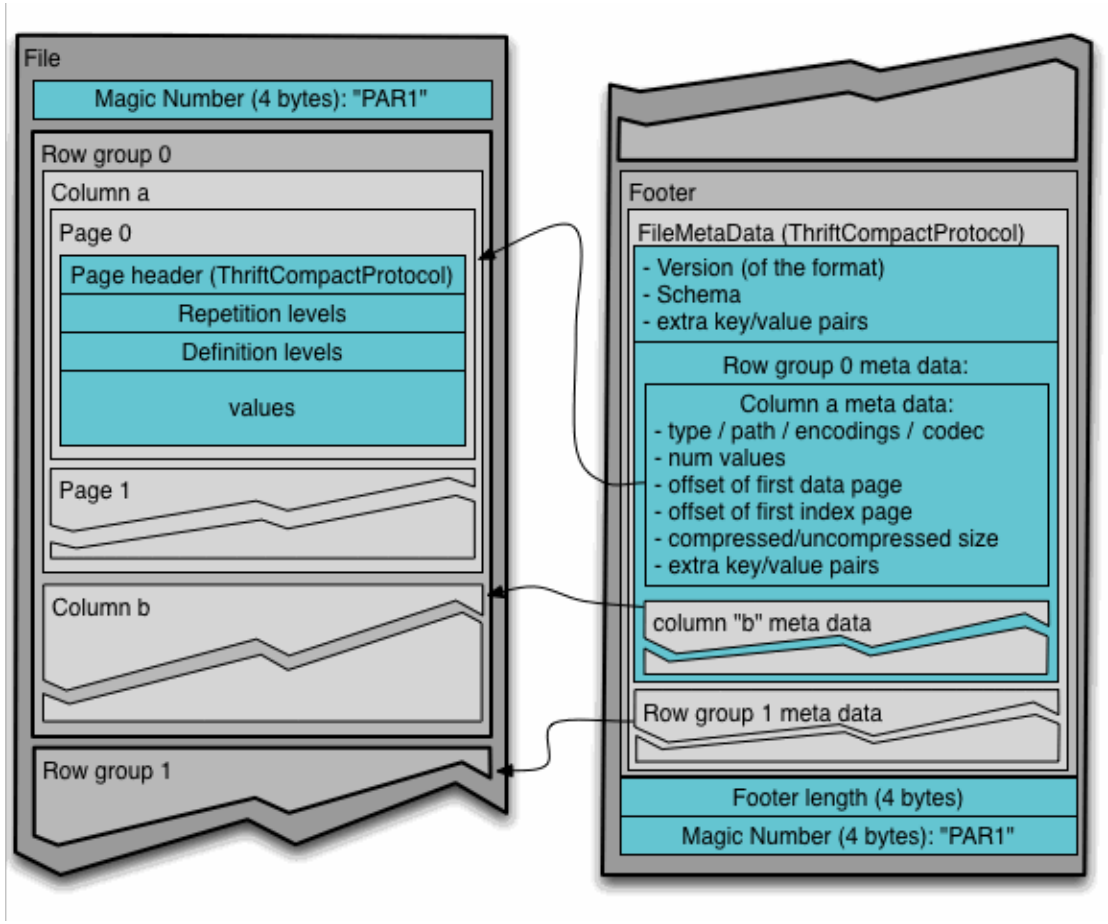


Figure from CMU 15-445/645

Storage Models (in a nutshell)

- Partition Attributes Across (PAX) model
  - Hybrid model
    - Horizontally partition data into **row groups**
    - Vertically partition attributes in each row group into **column chunks**
    - Still gives us row-wise locality when we need to reconstruct rows

- De-facto format for ML: ORC, Parquet, Arrow, Nimble, Vortex



Co-locating Compute and Data

Co-locating Compute and Data

Moving 1 PB over a 10 Gbps link

- Wire time: ~3 weeks.
- Egress cost: ~$90K at $0.09/GB every time.
- Engineering time: 6–9 months of pipeline re-validation, schema migration, and lineage rework.

Scale	The right architecture
GB	Move the data to the compute
TB	Co-locate storage and compute
PB	Lakehouse: bring Spark to the storage
PB+, many orgs	Federated / mesh: do not centralize at all

- Storage decisions from the last few slides are hard to reverse — data acquires “gravity” as consumers accumulate around it
- Practical consequence: choose the format, frameworks, and the location before the dataset is large

Recap: The Training Data Pipeline

ML Preprocessing

- **"Last-mile" of data processing**
  - f: batch of training samples → tensors in GPU HBM
- **Highly domain-specific, expressed as Python modules**
  - Vision: color augmentation, shear, contrast, … (e.g. torchvision)
  - Text: tokenize, normalize, pad, embed, … (e.g. tf.text)
  - Recommendation: hashing, bucketization, sparse feature assembly
- **Unique semantics: random augmentations, data ordering, epoch boundaries**

Unlike “offline” prep, this code lives inside the training job. Written by the modeler, “streaming” (no materialization), and rarely treated as infrastructure.

DataLoaders

```python
import os
import pandas as pd
from torchvision.io import decode_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = decode_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

- **DataLoader abstraction**
  - Map-style __getitem__(): random access to any sample by index (why is this bad?)
  - Iterable-style __iter__(): iterable over data samples (why might this be bad?)
  - Per-sample transforms, applied lazily

__getitem__ (or __iter__()) is called millions of times per epoch, in Python, and hides storage I/O, decode, and many tensor ops.

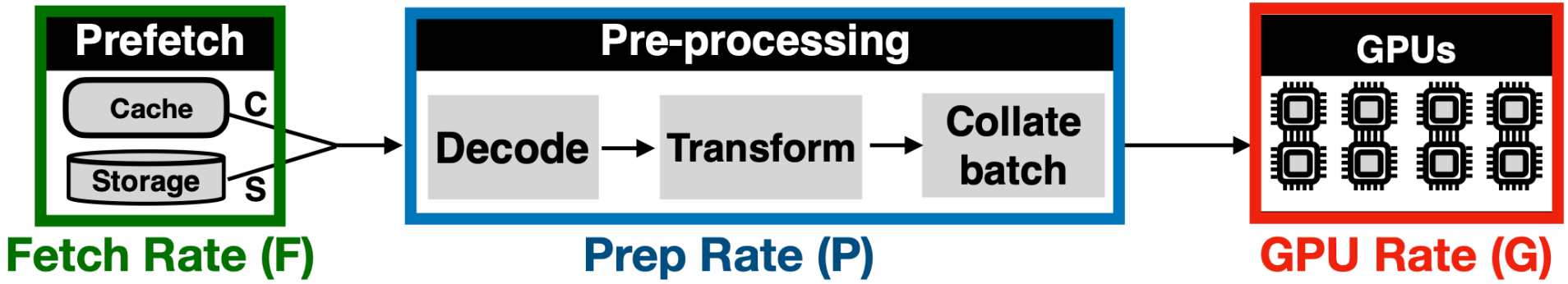
The ML Storage Hierarchy

Tier	Storage	Bandwidth	Cliff vs tier above
0	GPU HBM	3.35 TB/s	—
1	Host DRAM	200 GB/s	17x
2	Local NVMe	7–25 GB/s	10–30x
3	Parallel FS	4 GB/s / node	10x
4	Object storage	~1 GB/s / node	4x

- ~500x gap between GPU HBM and local NVMe!
- HBM can empty in 24ms, but it takes 400ms to refill from DRAM
  - The next batch must *already* be in HBM before the current one finishes.
  - Q: What happens if the batch is not ready yet?

Implication: Each tier is a prefetch buffer for the tier above it. Need to size the storage hierarchy so the entire storage path can feed the tier above it.

Data Stalls



A data stall occurs if G > min(F, P)

- F — fetch rate: reading bytes from cache or storage into host memory
- P — prep rate: decode, transform, collate into a batched tensor
- G — GPU rate: how fast the accelerator can consume batches
- **The input pipeline is a throughput system, so either F or P can starve the GPU.**

The Pipeline Equation

$$BW_{required} = N_{GPUs} \times U_{target} \times \left(S_{batch} / T_{iteration} \right)$$

Variable	Meaning	ImageNet on 256 GPUs
N_{GPUs}	Number of accelerators	256
U_{target}	Target utilization (0.8–0.95)	0.80
S_{batch}	Local batch size in bytes	256 × 150 KB
$T_{iteration}$	Forward–backward time	0.2 s
$BW_{required}$	**What the data pipeline must sustain**	39.3 GB/s, continuously throughout the training job

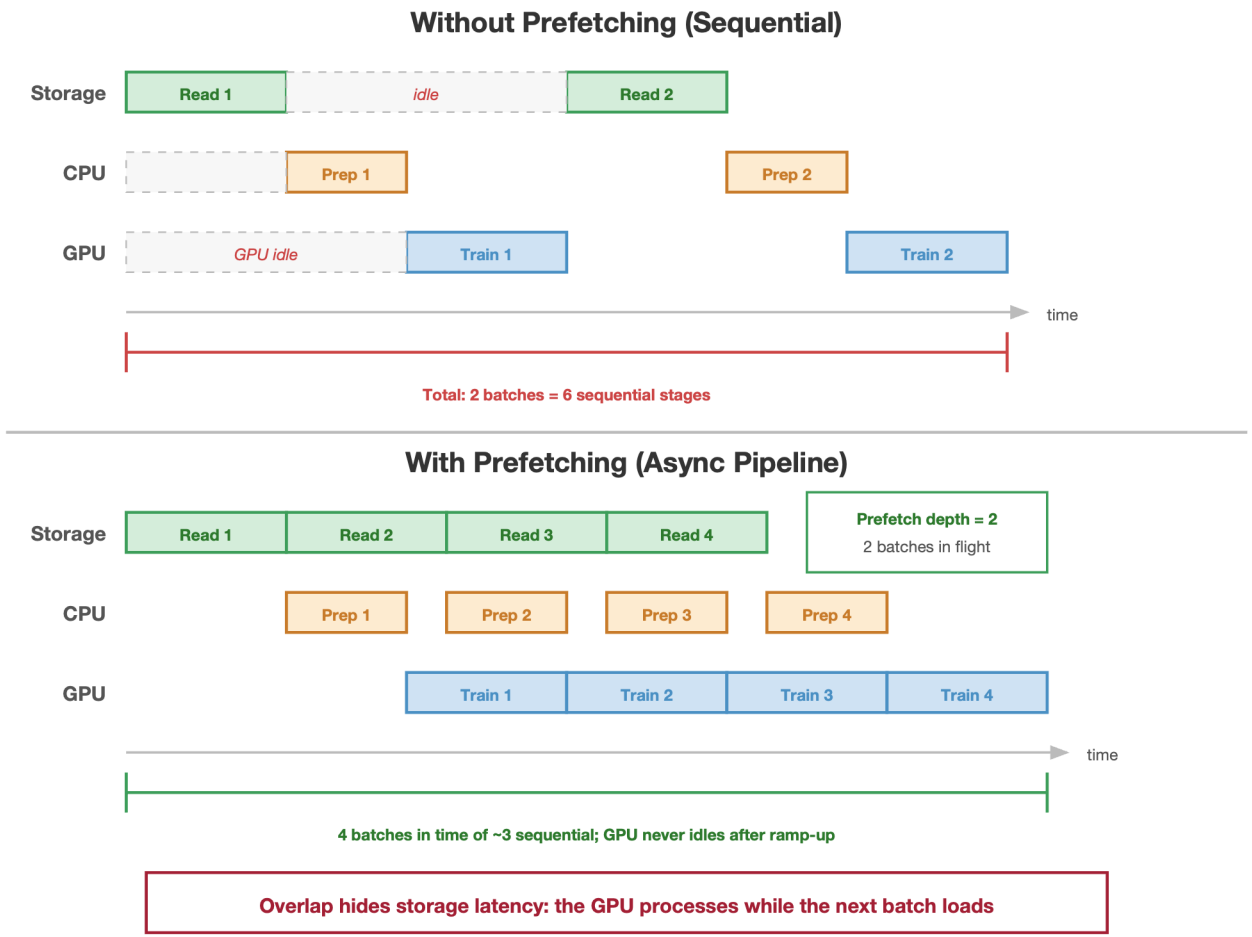
Data Stall Ratio

$$\text{Data stall \%} = (T_{step} - T_{compute}) / T_{step} \times 100$$

Stall ratio	Diagnosis	Action
< 2%	Healthy pipeline	Monitor. Do not over-invest here
2–10%	Marginal — will break at scale	Audit the pipeline, add prefetch depth
> 10%	Clear storage bottleneck	Redesign the data path

- **A 5% stall on 1,000 GPUs at $2/GPU-hour wastes about $16,800 in a single week. 1% is massive at scale.**
- **Consequence: Upgrading A100 to H100 without touching storage increases the stall ratio. Need to re-think storage architecture alongside accelerator upgrades.**

Mitigation 1: Pipelining and Prefetching



- Example:
  - T_compute = 200ms
  - T_I/O = 250ms
- Stall% without pipelining?
- Stall% with?

https://mlsysbook.ai/vol2/data_storage/data_storage.html

Mitigation 1: Pipelining and Prefetching

$$Q_{min} = \text{ceil}(T_{io,p99} / T_{compute})$$

Given	Value	Prefetch depth needed
T_compute per batch	200 ms	—
T_io average	250 ms	ceil(250/200) = 2 batches
T_io p99 (tail)	500 ms	ceil(500/200) = 3 batches

- **Size the buffer for P99, not the average**
  - Object storage makes this worse: P99 latencies of 500+ ms are routine

- The cost of depth is host memory. Q_min batches of tensors must competes with everything else on the node.

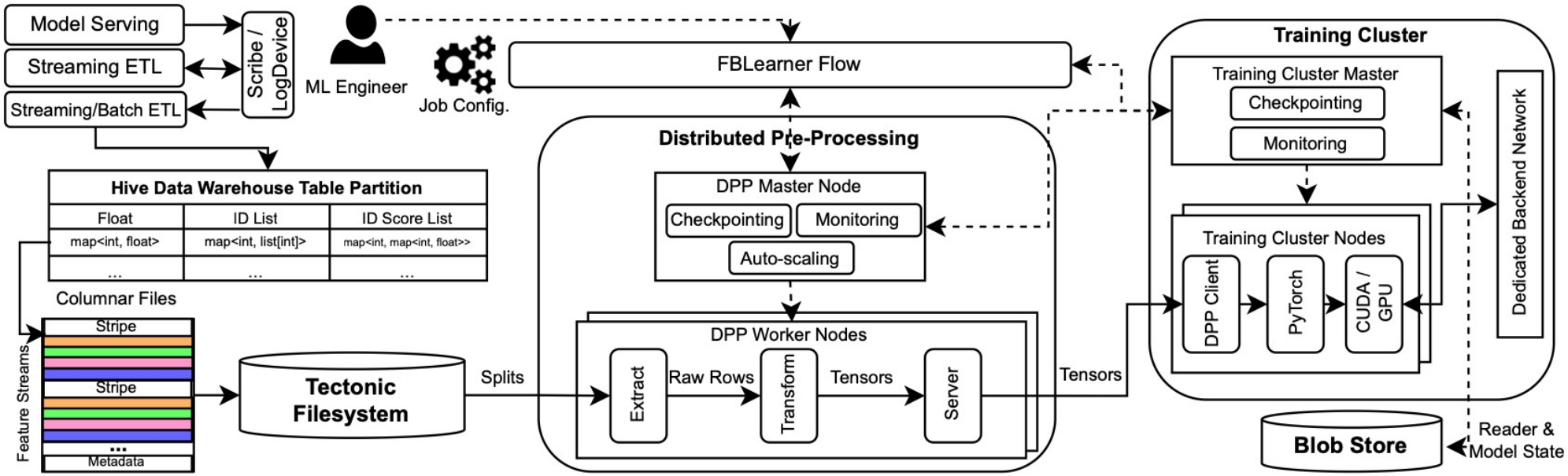
Mitigation 2: Scale-up within the node

Multi-process workers
Use multiple Python worker processes (num_workers) to overlap loading with compute and sidestep the GIL. Helps when P is the bottleneck. Costs host memory and CPU contention.

Pinned memory
Land tensors in physical, non-swappable pages so the GPU can DMA directly. Helps when P is the bottleneck by removing a CPU-side copy.

- How to diagnose if G, F or P is the bottleneck?
- Try these first, easy configuration change without much cost
- Downside of these options?

Mitigation 3: Scale-out and disaggregate



- Run preprocessing (and storage) as a dedicated, independently scaled service. Ship finished tensors to the trainer
  - Helps when P (disagg compute) or F (disagg storage) is the bottleneck, and breaks the fixed disk:CPU:GPU ratio of the training node

Mitigation 3: Change the graph

Caching / materialization

Write (partially) preprocessed data back to storage and reuse it across epochs and jobs. Helps P, and helps F when the materialized form is smaller. Constraint: you cannot materialize past a random augmentation.

Storage optimizations

Change placement (remote → local SSD, or into page cache) or format (row → columnar, or large sequential shards). Helps F.

Operator fusion/kernels

Combine adjacent operators to avoid materializing intermediates and keep data in cache. Helps P.

Reordering

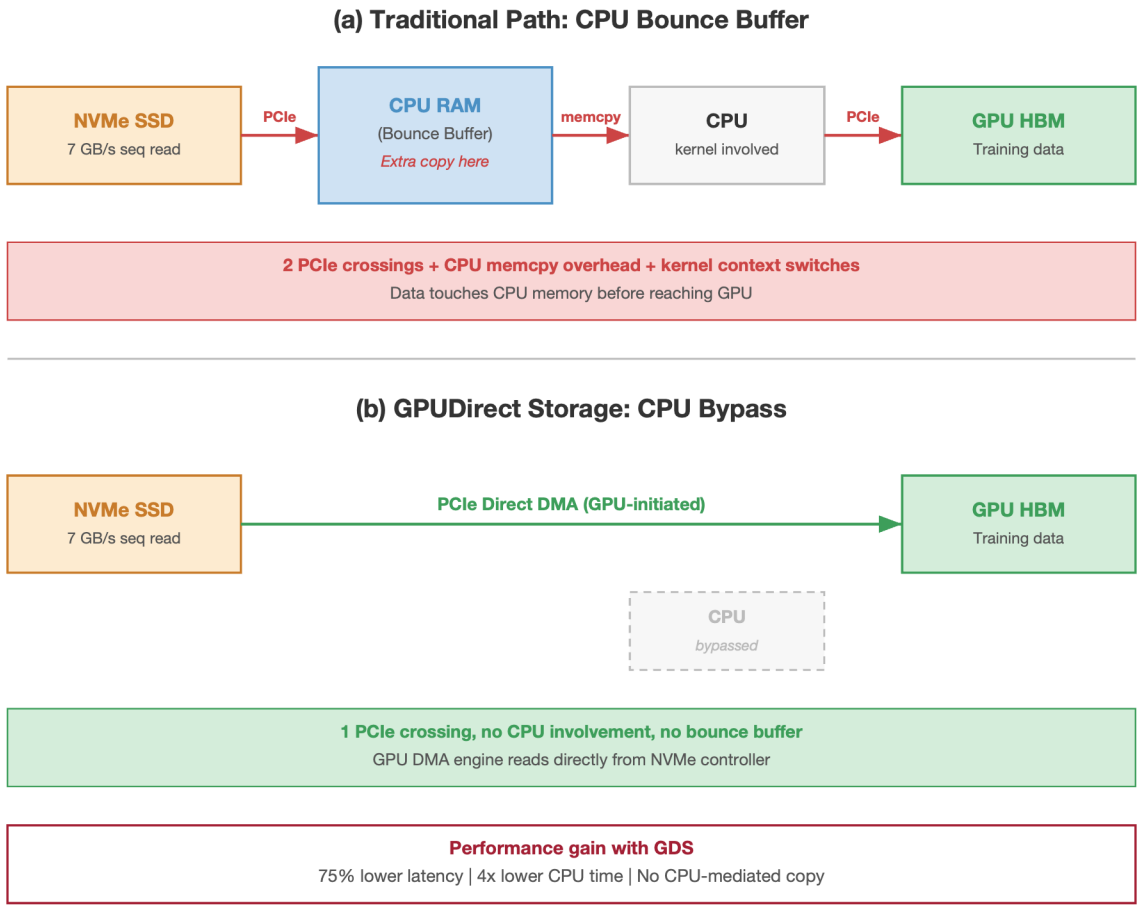
Change operator order to do less work, without changing semantics. Helps P.

Mitigation 3: Change the graph

- Question: When should we run a transformation “offline” versus “online”?

Mitigation 4: Offloading

- Use a different implementation, or different hardware, for the same operator
  - Vectorized CPU kernels (AVX), GPU decode and augmentation (e.g. DALI, nvJPEG), dedicated decoders.
- Helps P — but can compete for GPU resources
  - Offloading augmentation to the GPU relieves the CPU and consumes GPU cycles. It works only when the GPU has spare cycles.



https://mlsysbook.ai/vol2/data_storage/data_storage.html

Summary

- Measure first: Figure out if G, P, or F is your bottleneck
- Move work in space
  - Scale-up, scale-out, offload, use GDS
- Move work in time
  - Cache/materialize intermediate values, prefetch
- Move work in order
  - Modify your graph to reduce work – move ops between online/offline, fuse, or reorder ops

Changing Storage Bottlenecks

175B-parameter model, 30-day run, 256 nodes	Volume
Training dataset (compressed)	3 TB
Model weights (FP16)	350 GB
Optimizer state (FP32)	1,400 GB
A single full checkpoint	1,750 GB
All checkpoints, 30 days at 10-minute intervals	~7.6 PB

The inversion: checkpoint I/O (~7.6 PB written) is much larger than training-data reads (3 TB read). For frontier-scale models*, checkpointing, not data loading, is the dominant storage workload.

- It competes for the same bandwidth your input pipeline needs
- It is also bursty: every worker writes simultaneously, so the file system must be provisioned for peak, not average
  - The standard mitigation is staging — write to local NVMe fast, replicate to the parallel FS in the background.

*LLMs, the story is different for models which train on non-text inputs which require large datasets (e.g., images, videos)