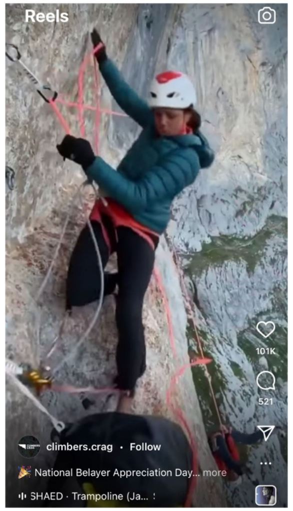
RecD: Deduplication for End-to-End Deep Learning Recommendation Model Training Infrastructure

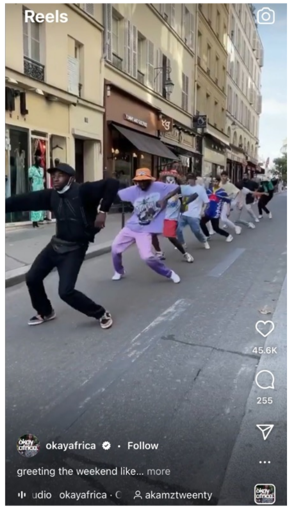
Mark Zhao[*], Dhruv Choudhary, Devashish Tyagi, Ajay Somani, Max Kaplan, Sung-Han Lin, Sarunya Pumma, Jongsoo Park, Aarti Basant, Niket Agarwal, Carole-Jean Wu, and Christos Kozyrakis[*]

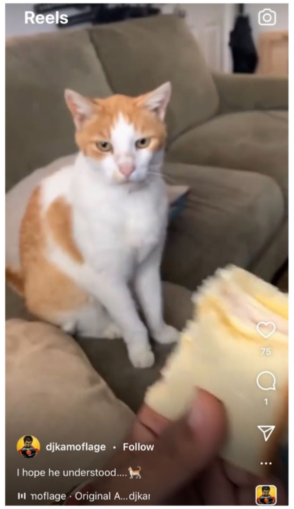
[*]Stanford University, Meta

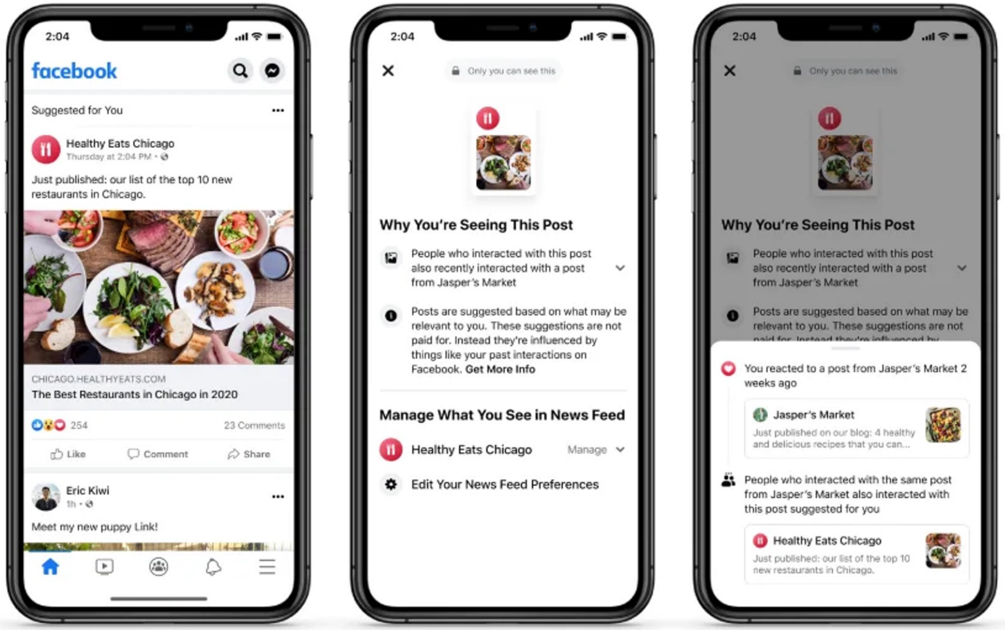
Sixth Conference on Machine Learning and Systems (MLSys 2023)

Machine Learning at Meta



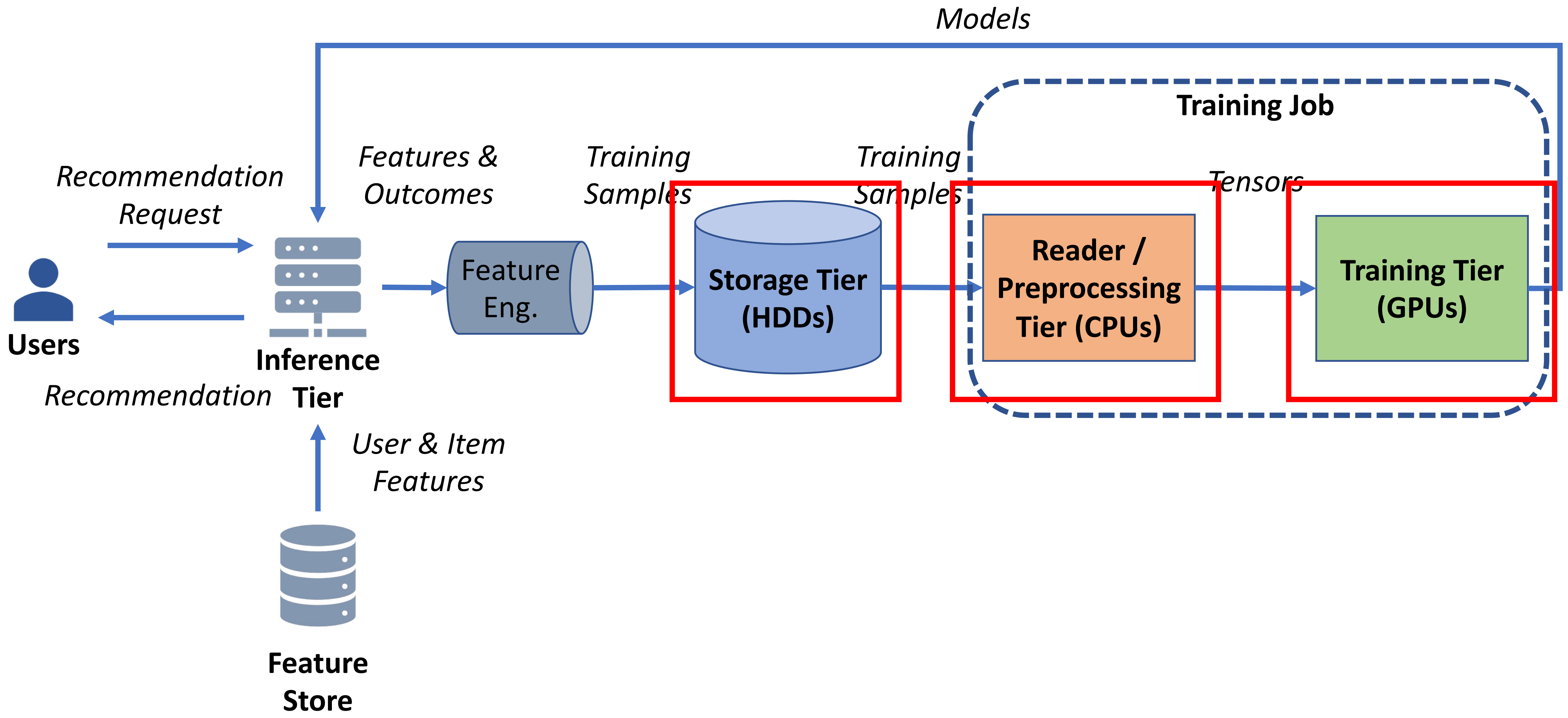






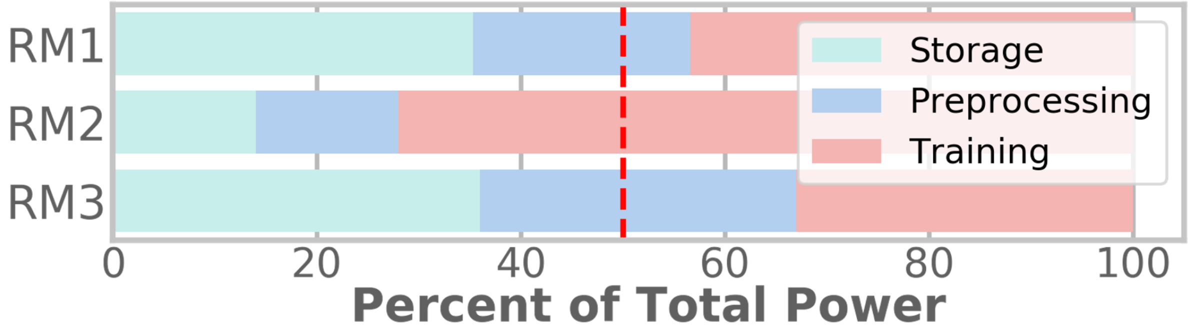
Deep Learning Recommendation Models (DLRMs) are business-critical and dominate AI training demand

End-to-End DLRM training infrastructure



End-to-end infrastructure optimization

Storage, *preprocessing*, and *training* each require immense infrastructure resources



Our approach: **Co-design** efficiency optimizations across the **end-to-end pipeline** to continue scaling ML systems

M. Zhao, et al., Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training, ISCA'22

Understanding DLRM datasets

Sparse Feature:
Last N liked post IDs

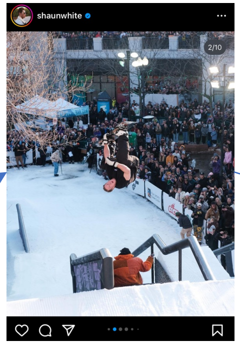
Inference Tier

DLRM Dataset Table

Features *map<feature_id: value>*	Label *int*

Alice

[31, 55, 17, …]



No Like

last_n_liked: [31, 55, 17, …]	0

[31, 55, 17, …]

No Like

last_n_liked: [31, 55, 17, …]	0
last_n_liked: [31, 55, 17, …]	0

[31, 55, 17, …]

Like

last_n_liked: [31, 55, 17, …]	0
last_n_liked: [31, 55, 17, …]	0
last_n_liked: [31, 55, 17, …]	1

[42, 31, 55, 17, …]

Understanding DLRM datasets

DLRM Dataset Table

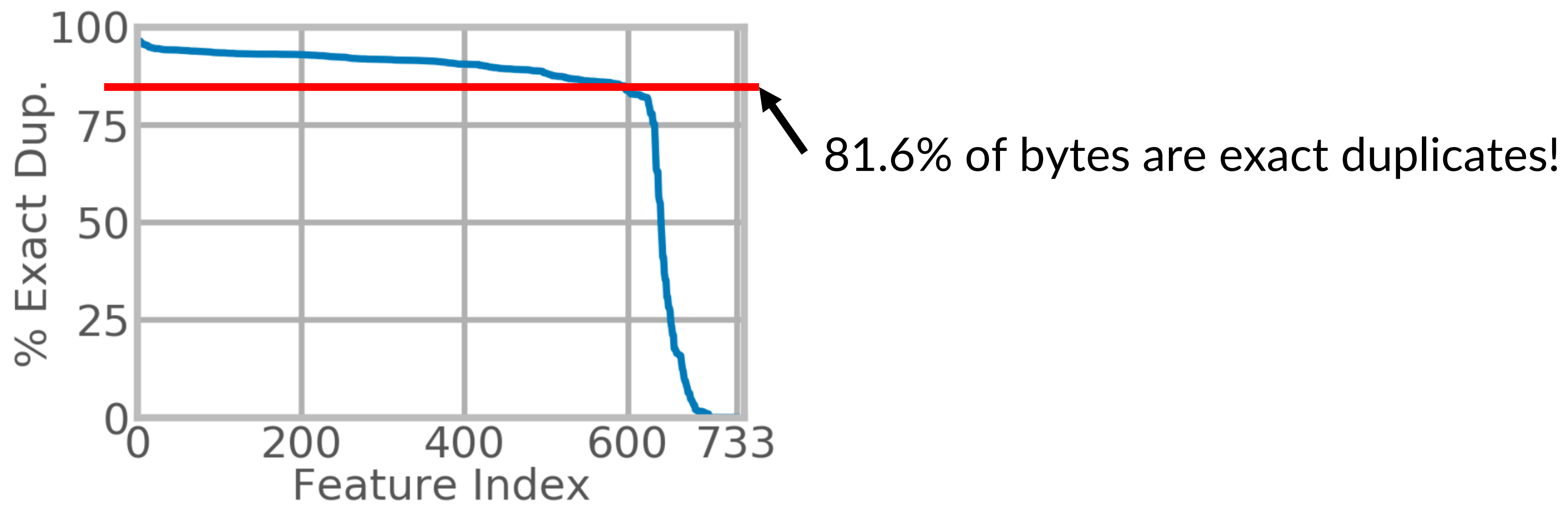
Features *map<feature_id: value>*	Label *int*
last_n_liked: [31, 55, 17, …]	0
last_n_liked: [31, 55, 17, …]	0
last_n_liked: [31, 55, 17, …]	1

Many other sparse features e.g., {comment/share/post} history, device type, etc. →

Intuition: Many sparse features are infrequently updated across a user's samples, resulting in high duplication

Understanding DLRM datasets: % duplication

Intuition: Expensive sparse features largely duplicated across a user's samples

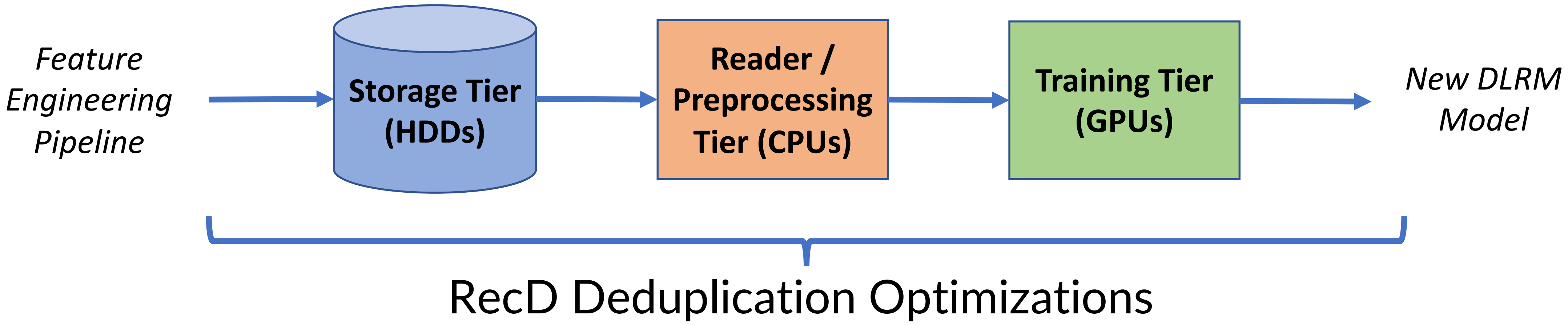


Opportunity:

Address overheads caused by duplicate sparse features via deduplication

RecD: End-to-end deduplication optimizations

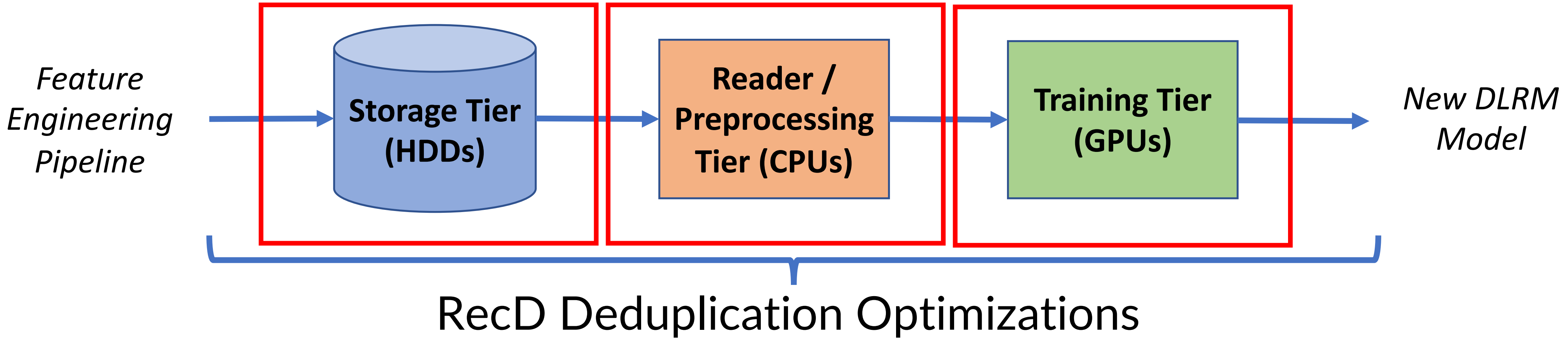
RecD improves **storage**, **preprocessing**, and **training** efficiency via deduplication



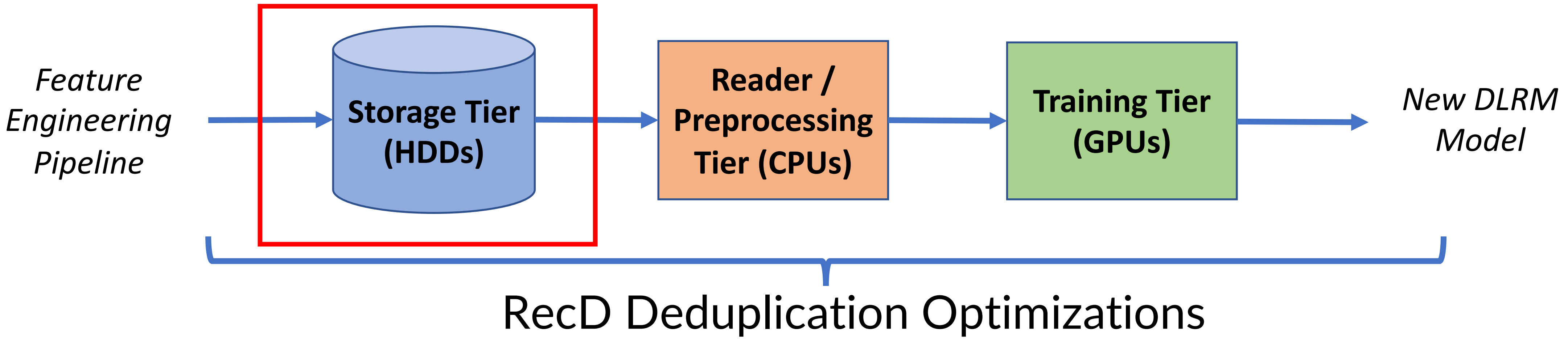
Key Insight: Upstream optimizations enable further downstream optimizations

RecD: End-to-end deduplication optimizations

- Storage
  - **Coalesce** duplicate samples to maximize deduplication potential
- Preprocessing
  - **Encode** InverseKeyedJaggedTensors (IKJTs) to deduplicate each batch
- Training
  - **Accelerate** DLRM training using IKJT-centered modules

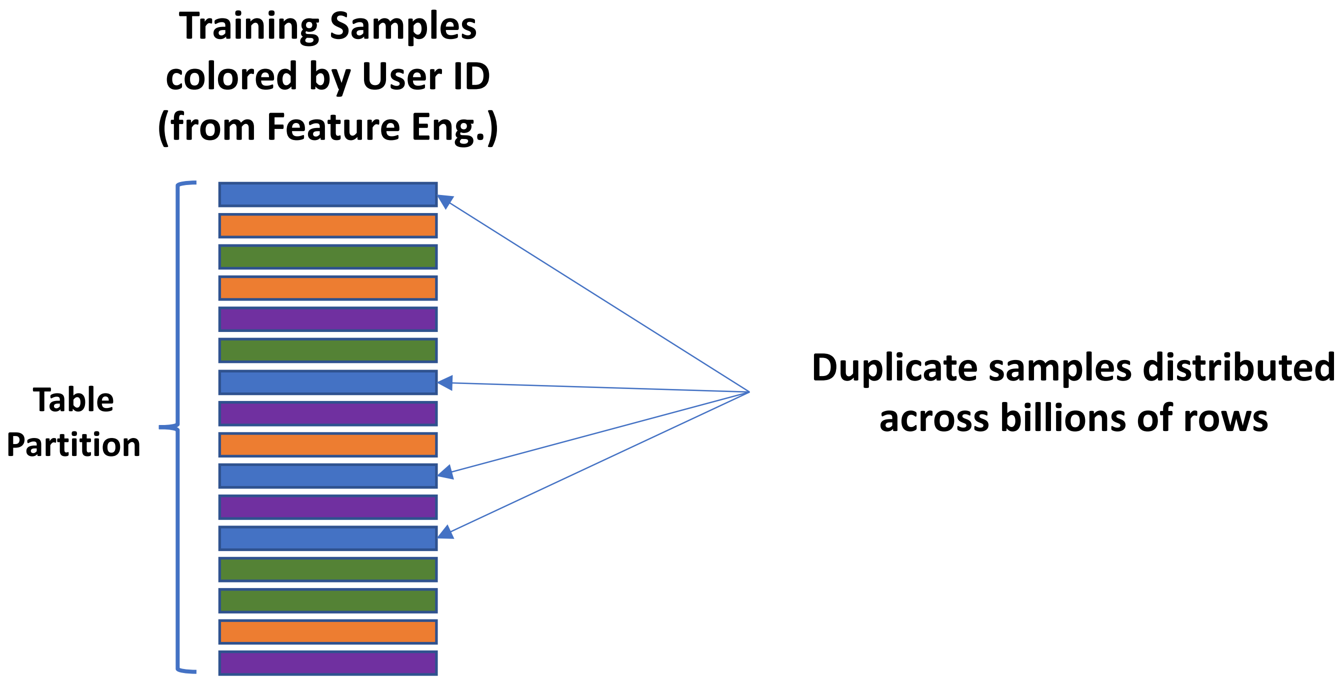


RecD: **Coalesce** DLRM training samples

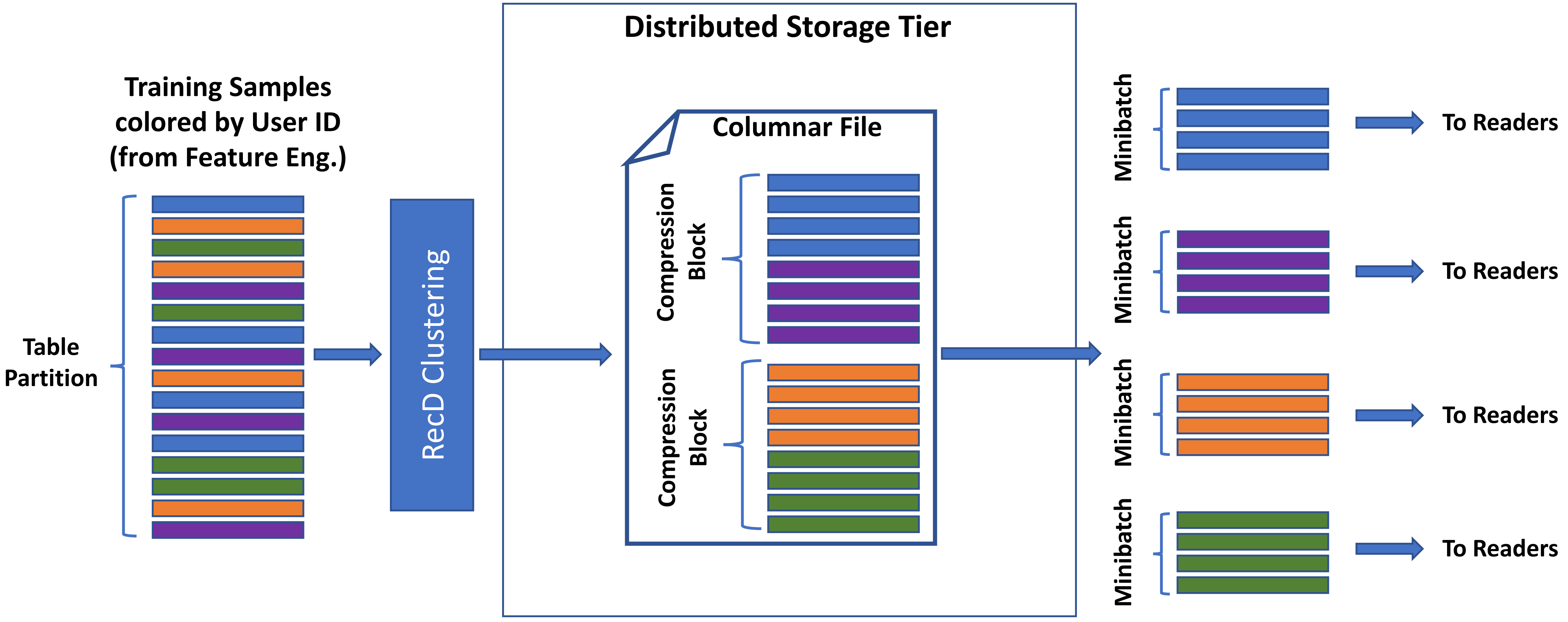


Challenge: How do we maximize deduplication *potential* of the entire pipeline?

RecD: **Coalesce** DLRM training samples

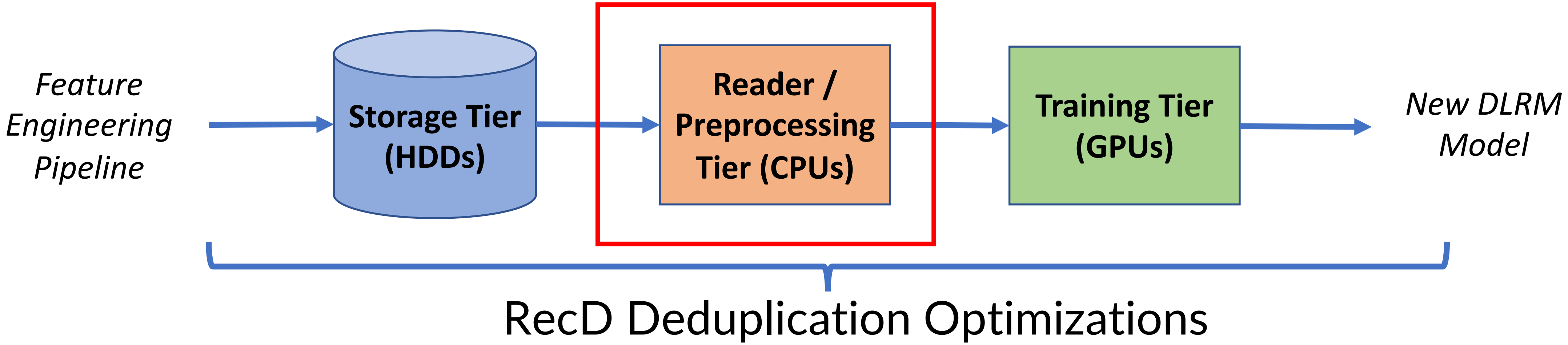


RecD: **Coalesce** DLRM training samples



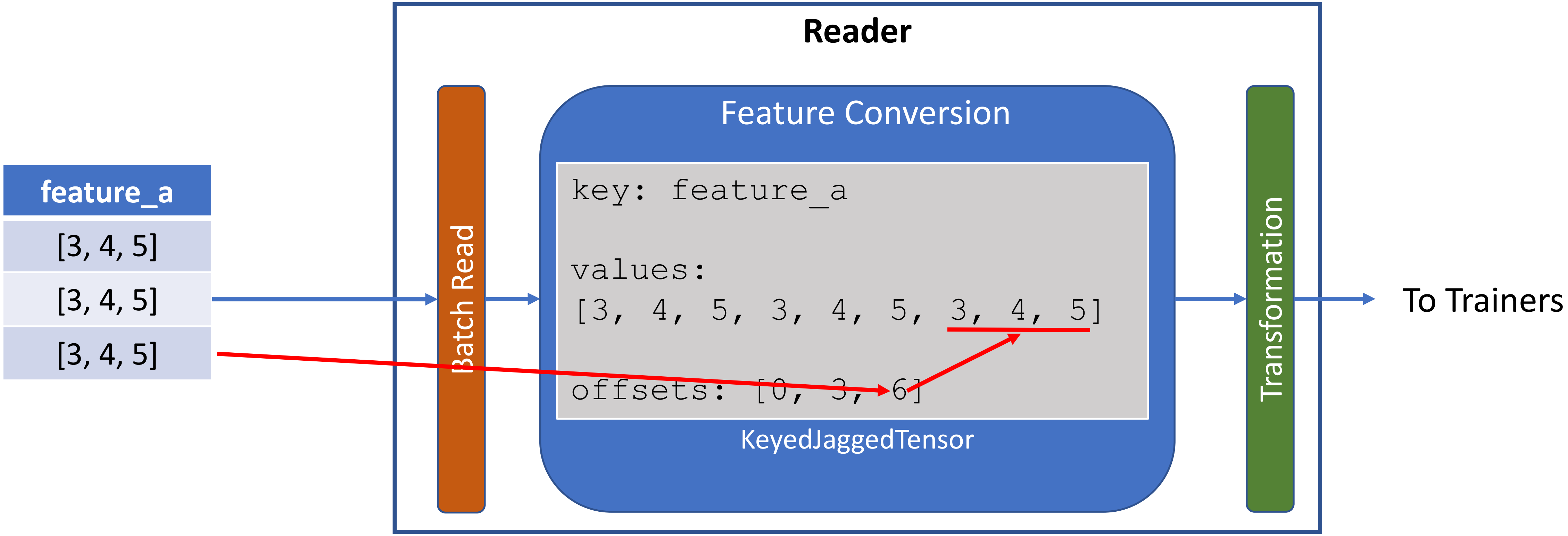
Maximize duplication to improve **storage** and **read I/O** efficiency via compression

RecD: **Encode** deduplicated InverseKeyedJaggedTensors

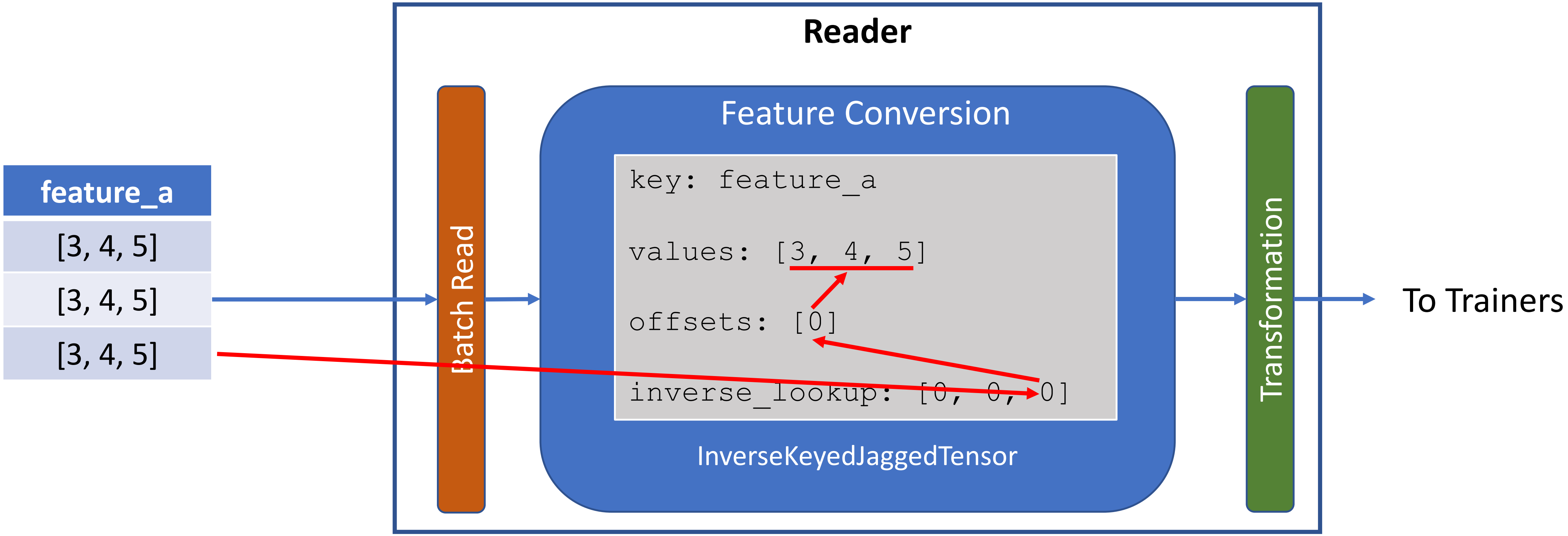


Challenge: How do we deduplicate downstream *tensor* operations?

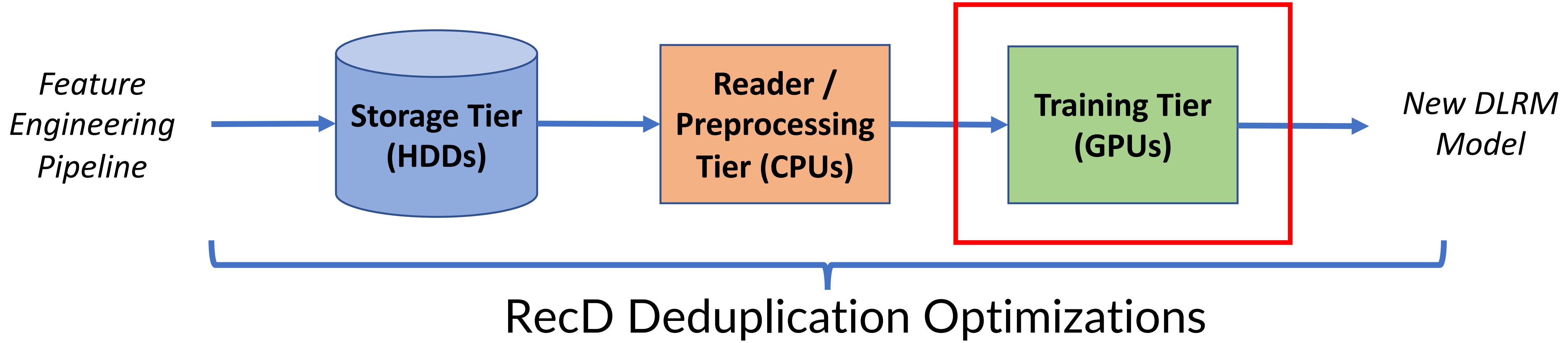
Background: PyTorch KeyedJaggedTensors



RecD: **Encode** InverseKeyedJaggedTensors



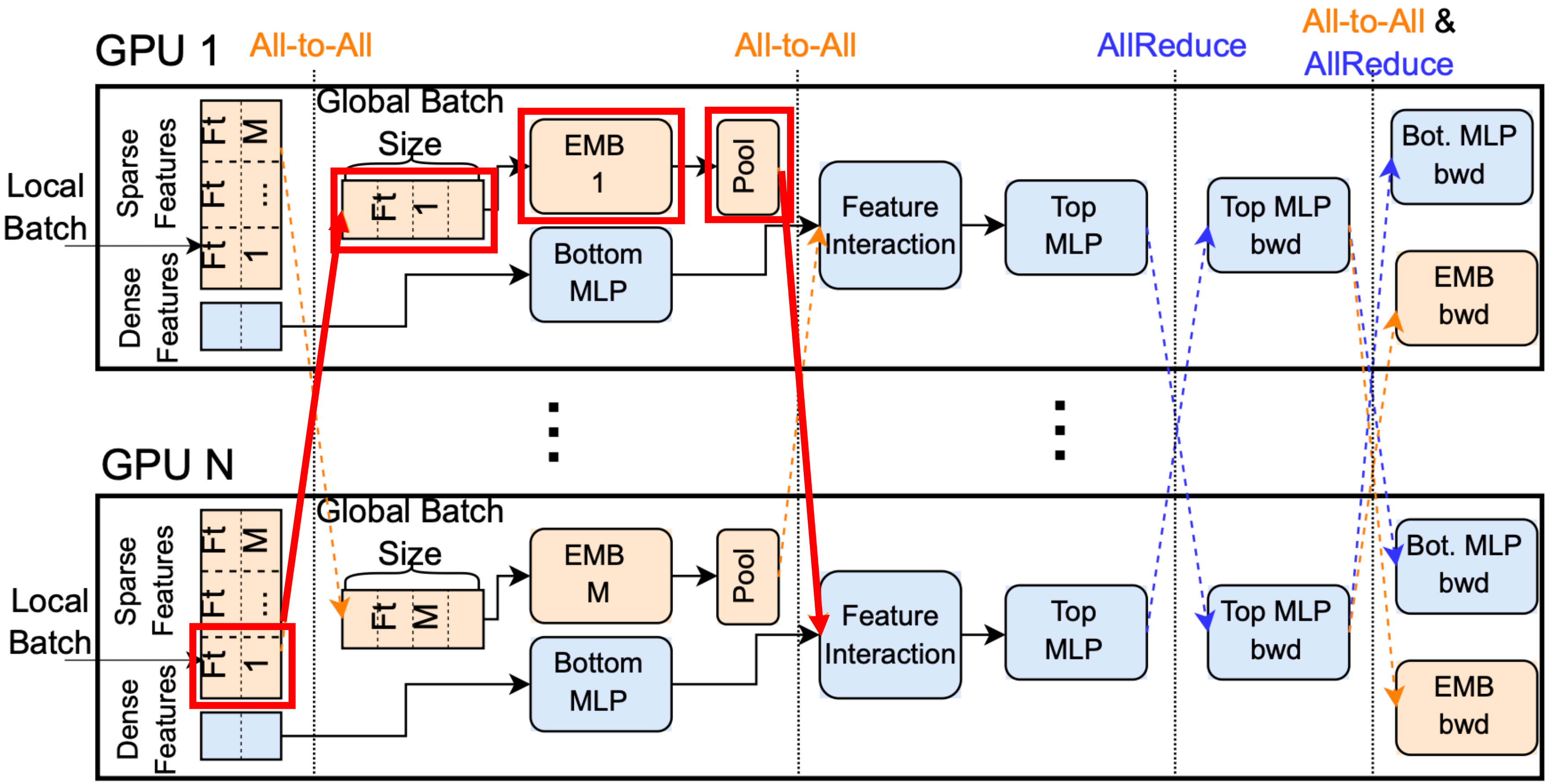
RecD: **Accelerate** DLRM training



Challenge: How do we leverage IKJTs to improve training throughput?

Background: DLRM training

Synchronous **model parallel** and **data parallel** training



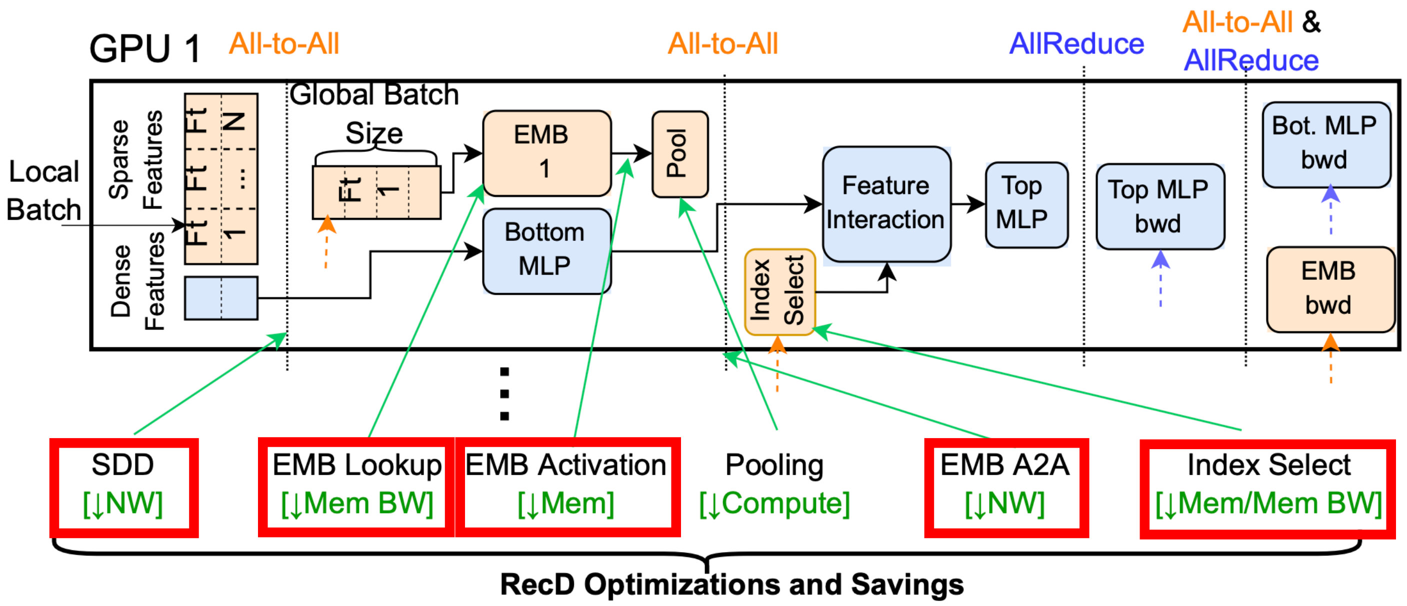
RecD: **Accelerate** DLRM training

Plug-and-play modules that operate on IKJTs

```
KJT: [3, 4, 5, 3, 4, 5, 3, 4, 5]
```

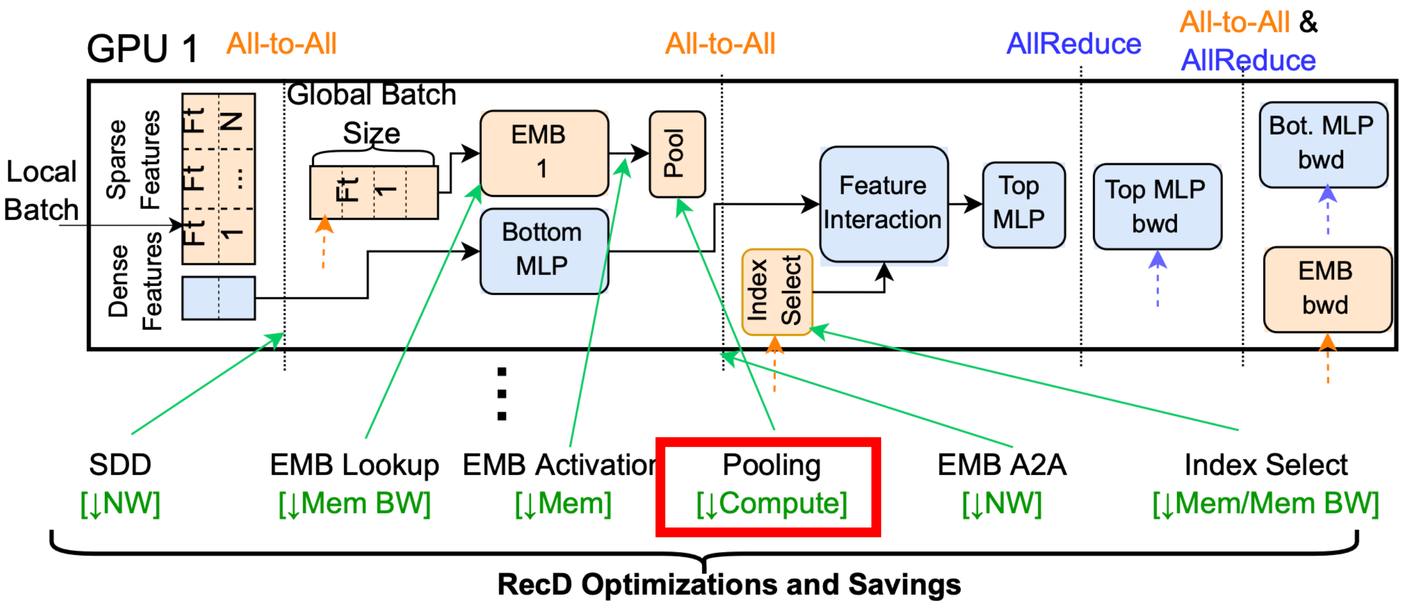
RecD

```
IKJT: [3, 4, 5]
```

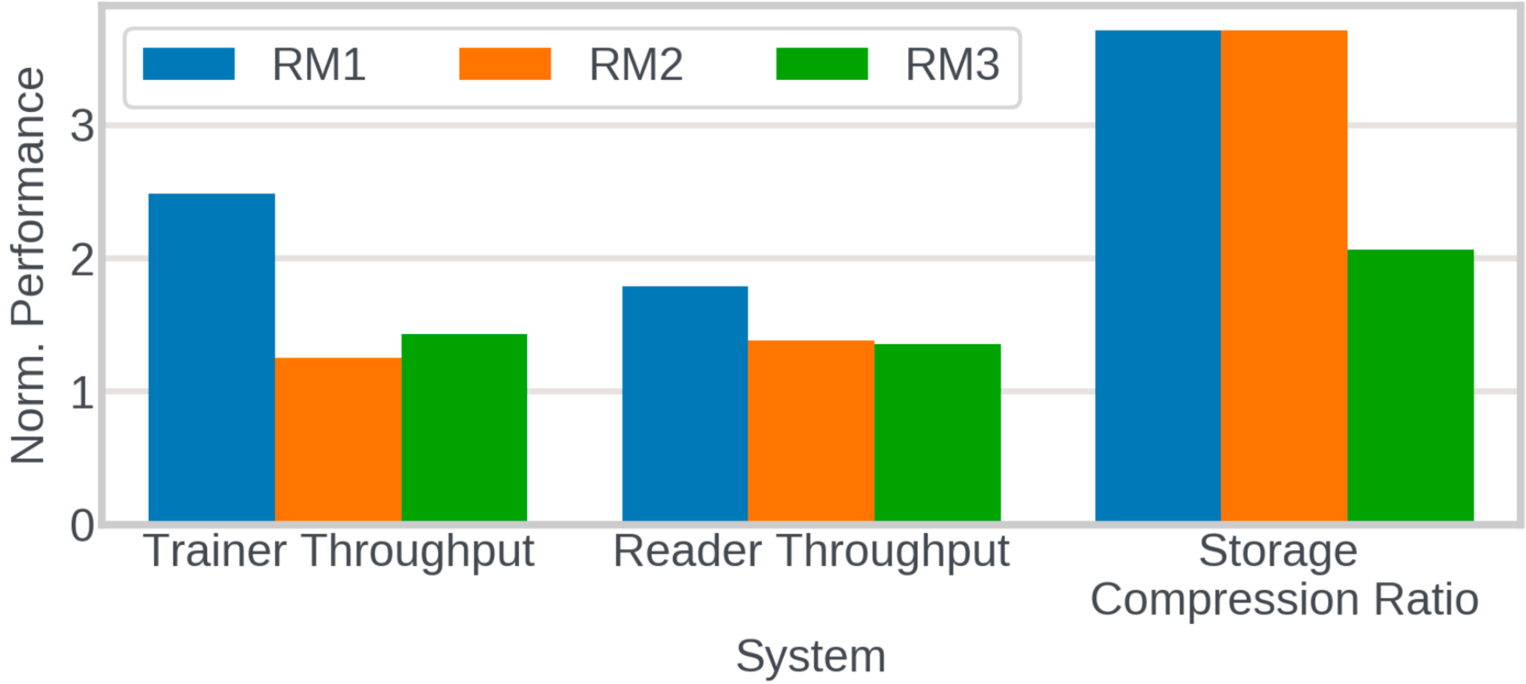


RecD: **Accelerate** DLRM training

feature_a
[3, 4, 5]
[3, 4, 5]
[3, 4, 5]



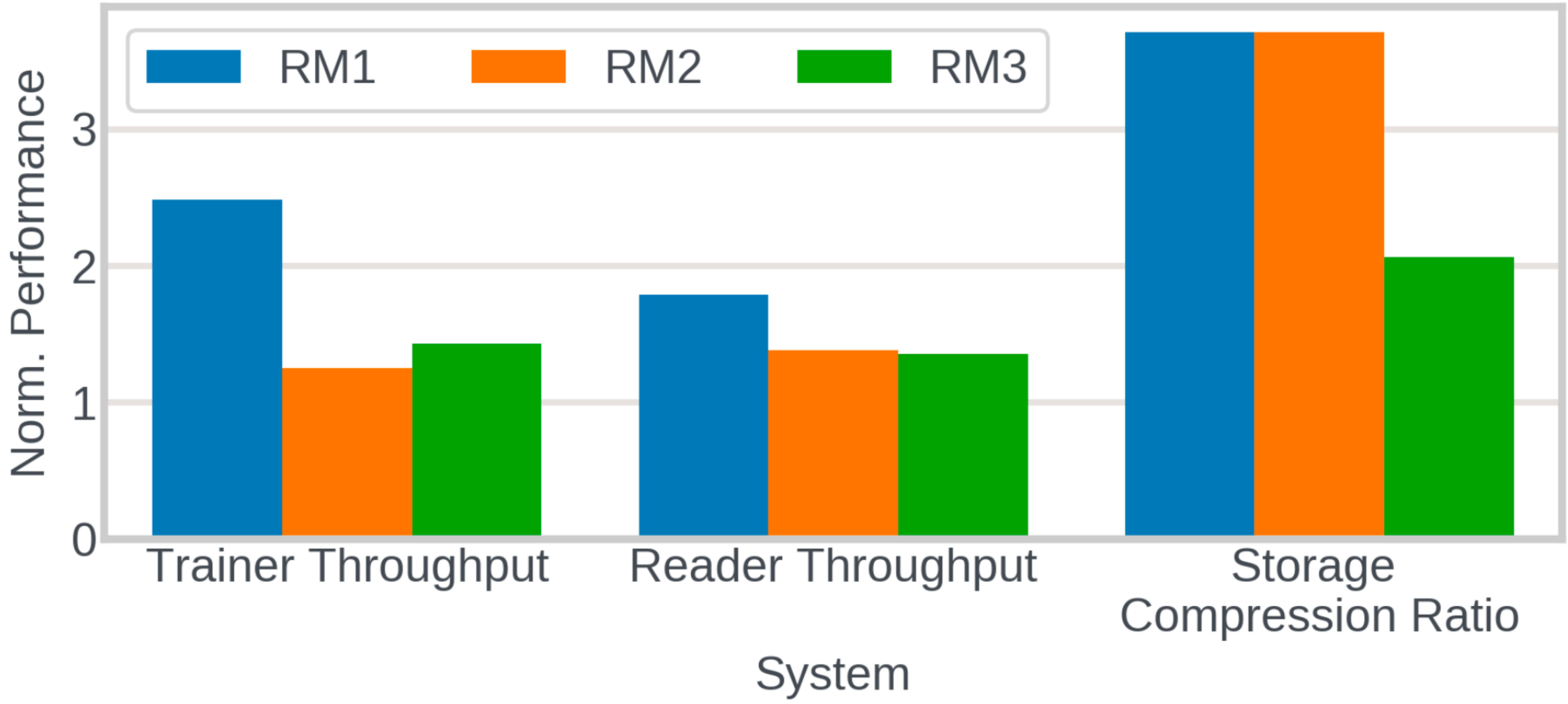
Results



RecD improves production training, preprocessing, and storage efficiency on average by 72%, 51%, and 216%, respectively.

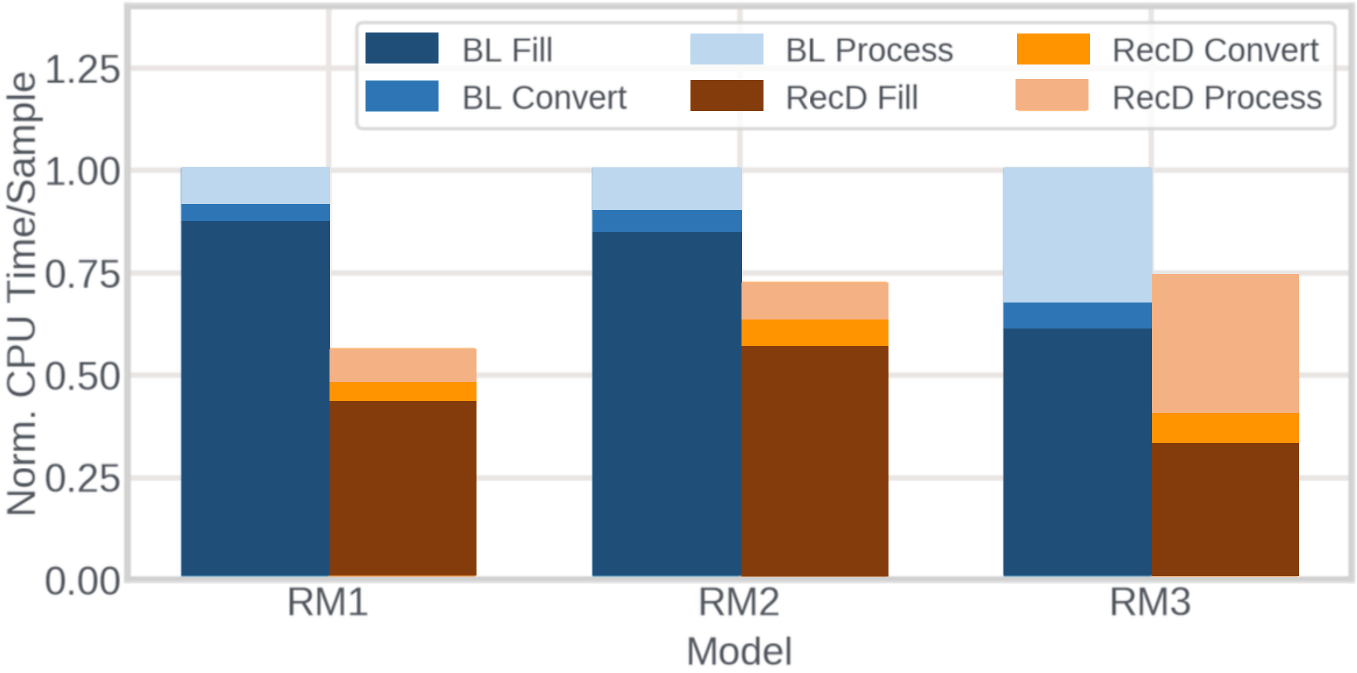
Storage efficiency

- Higher native compression ratios



Reader efficiency

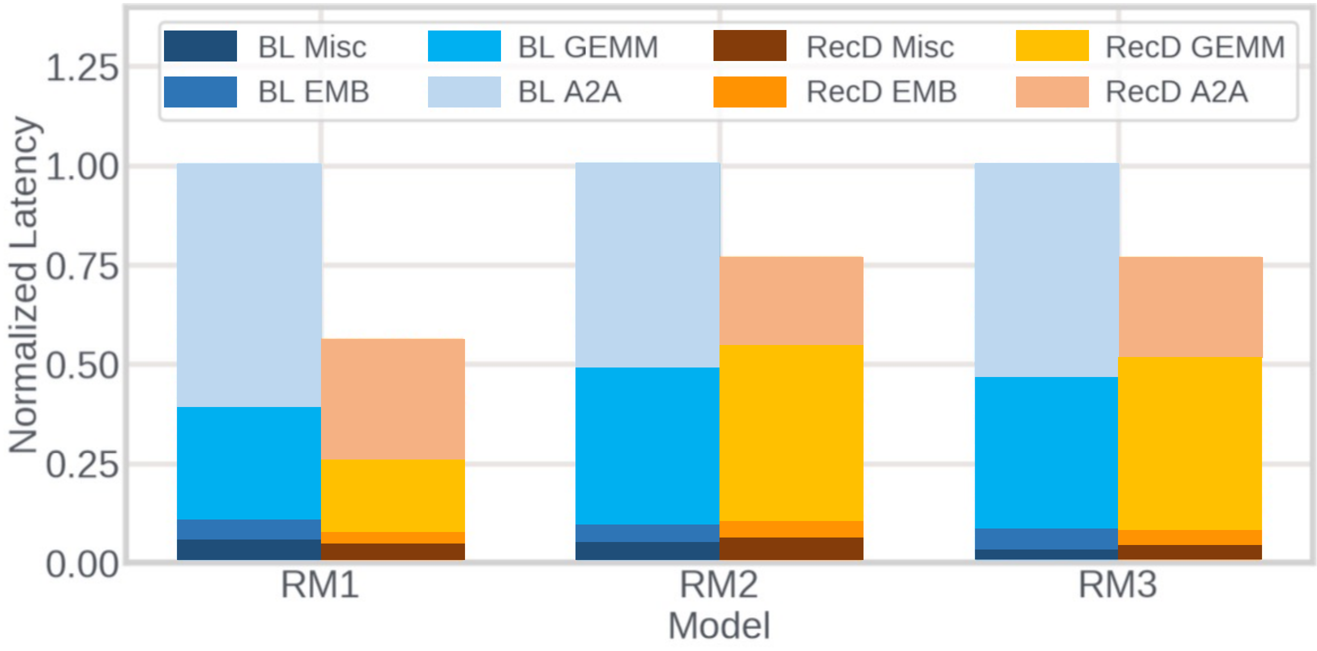
- Better storage compression for reading/extraction
- Eliminate redundant transformations



Reader CPU time/sample versus baseline

Trainer efficiency

- Smaller all-to-all data transfers
- Fewer GEMMs for pooling operations



Trainer iteration latency versus baseline

Summary

- DLRM datasets exhibit high sparse feature duplication, leading to massive inefficiencies in training pipelines.

- *RecD* is a suite of end-to-end deduplication optimizations targeting **storage**, **preprocessing**, and **training**.

- *RecD* improves training, preprocessing, and storage efficiency by **72%**, **51%**, and **216%**, respectively.

myzhao@cs.stanford.edu

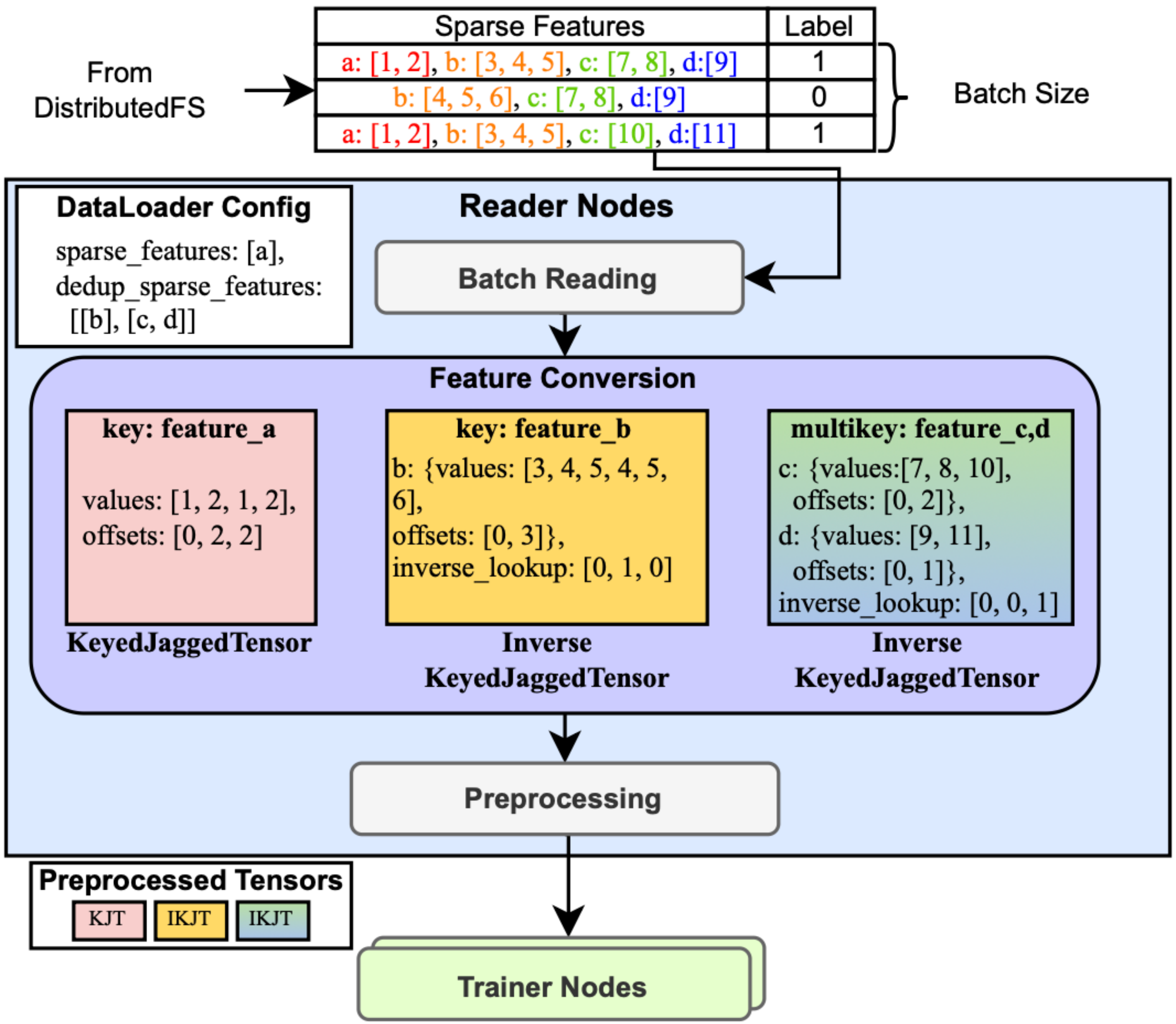
Meta

Scaling hyperparameters

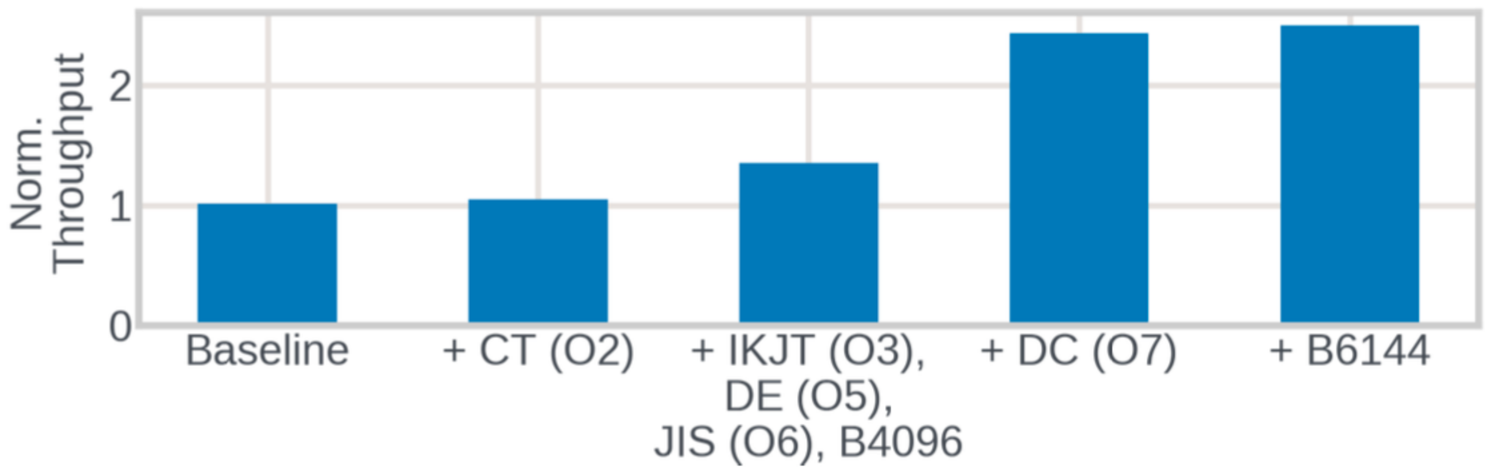
Config.	Norm. QPS	Max Mem. Util.	Avg. Mem. Util.	Norm. Comp. Efficiency (flop/s/GPU)
Baseline	1.00	99.90	72.83	1.00
RecD	1.89	27.76	22.20	1.73
RecD + EMB D256	1.55	40.87	31.17	1.92
RecD + B6144	2.26	91.78	51.55	2.12

RecD reduces GPU resource requirements, unlocking more complex models

Optimization	Target System	Benefit
O1: Log Sharding (§4.1)	LoggingService	Improves black-box compression ratios to reduce LoggingService network RX/TX and storage demands.
O2: Cluster by Session (§4.1)	ETL	Session sample co-location enables readers/trainers to exploit duplicate features. Improves file compression ratios, reducing storage and read IOPS demands.
O3: Inverse KJTs (§4.2)	Readers	New tensor encoding allows downstream preprocessing/training operations to use deduplicated features, enabling significant resource savings.
O4: Deduplicated Preproc. (§4.3)	Readers	IKJT preprocessing modules reduce preprocessing compute demands. Deduplicated outputs require less NW bandwidth between reader and trainers.
O5: Deduplicated EMB (§5)	Trainers	Reduced per-iteration trainer compute/memory/NW demands by deduplicating EMB features, lookups, and activations.
O6: JaggedIndexSelect (§5)	Trainers	Reduced memory copy overheads by enabling index select without first converting jagged tensors to a dense representation.
O7: Deduplicated Compute (§5)	Trainers	Reduced compute for sparse feature modules (esp. attention pooling) by allowing them to operate on deduplicated tensors.



Sparse Features
Label
a: [1, 2], b: [3, 4, 5], c: [7, 8], d:[9]
1
b: [4, 5, 6], c: [7, 8], d:[9]
0
a: [1, 2], b: [3, 4, 5], c: [10], d:[11]
1
From DistributedFS
Batch Size
DataLoader Config
sparse_features: [a],
dedup_sparse_features:
[[b], [c, d]]
Reader Nodes
Batch Reading
Feature Conversion
key: feature_a
values: [1, 2, 1, 2],
offsets: [0, 2, 2]
KeyedJaggedTensor
key: feature_b
b: {values: [3, 4, 5, 4, 5, 6],
offsets: [0, 3]},
inverse_lookup: [0, 1, 0]
Inverse KeyedJaggedTensor
multikey: feature_c,d
c: {values:[7, 8, 10],
offsets: [0, 2]},
d: {values: [9, 11],
offsets: [0, 1]},
inverse_lookup: [0, 0, 1]
Inverse KeyedJaggedTensor
Preprocessing
Preprocessed Tensors
KJT
IKJT
IKJT
Trainer Nodes


Norm.
Throughput
0
1
2
Baseline
+ CT (O2)
+ IKJT (O3),
DE (O5),
JIS (O6), B4096
+ DC (O7)
+ B6144

Experiment	Read Bytes (GB)	Send Bytes (GB)
Baseline	538	837
with Cluster	179	837
with IKJT	179	713

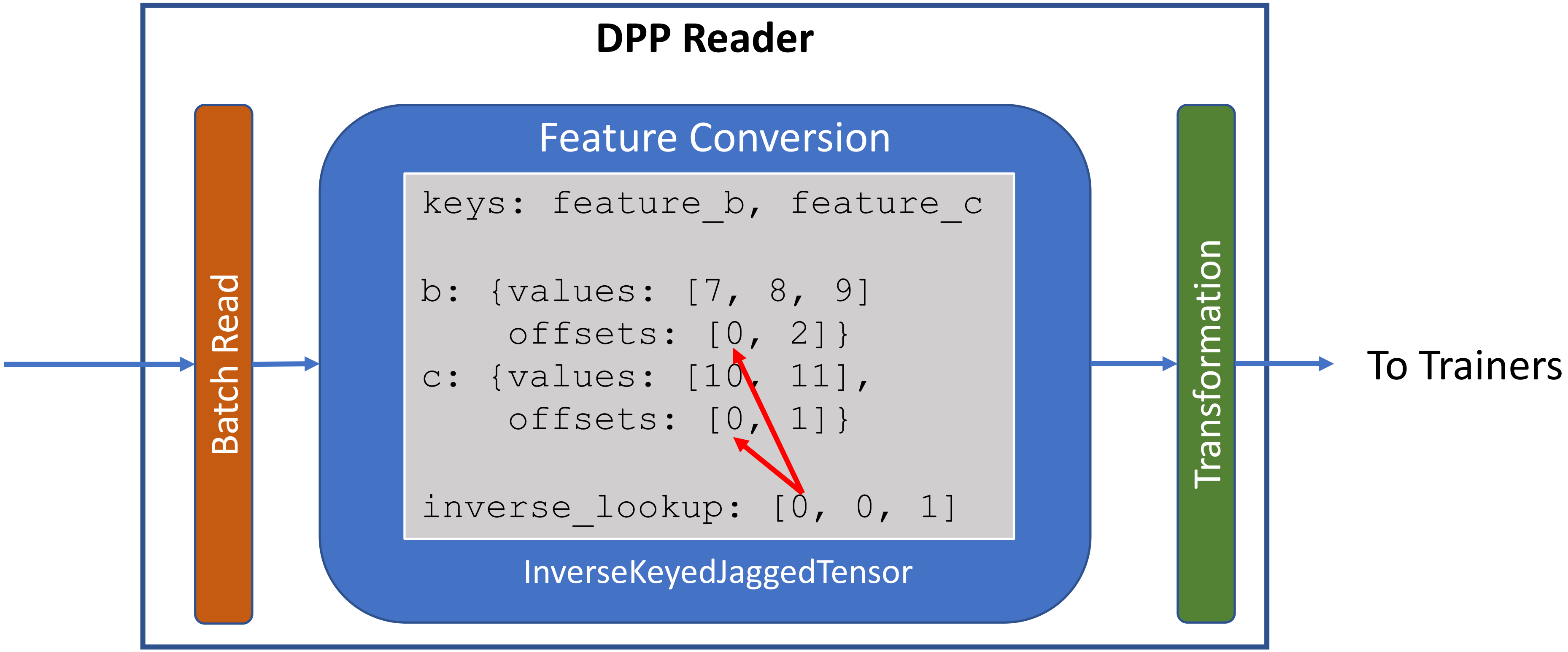
$$DedupeLen(f) = l(f) * B * (1 - (S - 1) * S^{-1} * d(f))$$
$$DedupeFactor(f) = l(f) * B / DedupLen(f)$$

Discussion

- Boosting deduplication factors
- Alternative Solutions
- Partial deduplication

RecD: **Encode** InverseKeyedJaggedTensors

feature_b	feature_c
[7, 8]	[10]
[7, 8]	[10]
[9]	[11]



Key deduplication challenges

- How do we **coalesce** duplicate samples into a training batch?
- When and how do we **encode** duplicate sparse features?
- How can we **exploit** deduplication to improve training throughput?