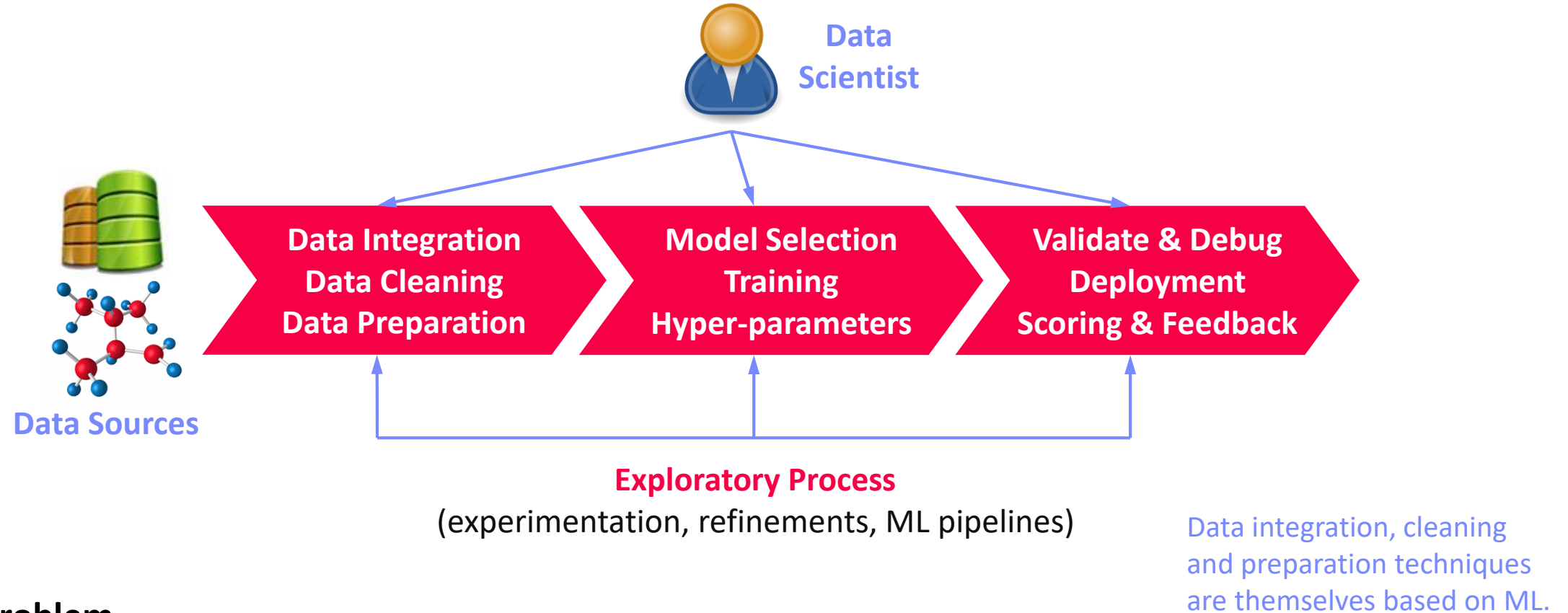


LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems

Arnab Phani¹, Benjamin Rath¹, Matthias Boehm¹

¹ Graz University of Technology; Graz, Austria

Exploratory Data Science



■ Problem

- High **computational redundancy** in ML pipelines
- **Reproducibility** and **explainability** of trained models (data, parameters, prep)

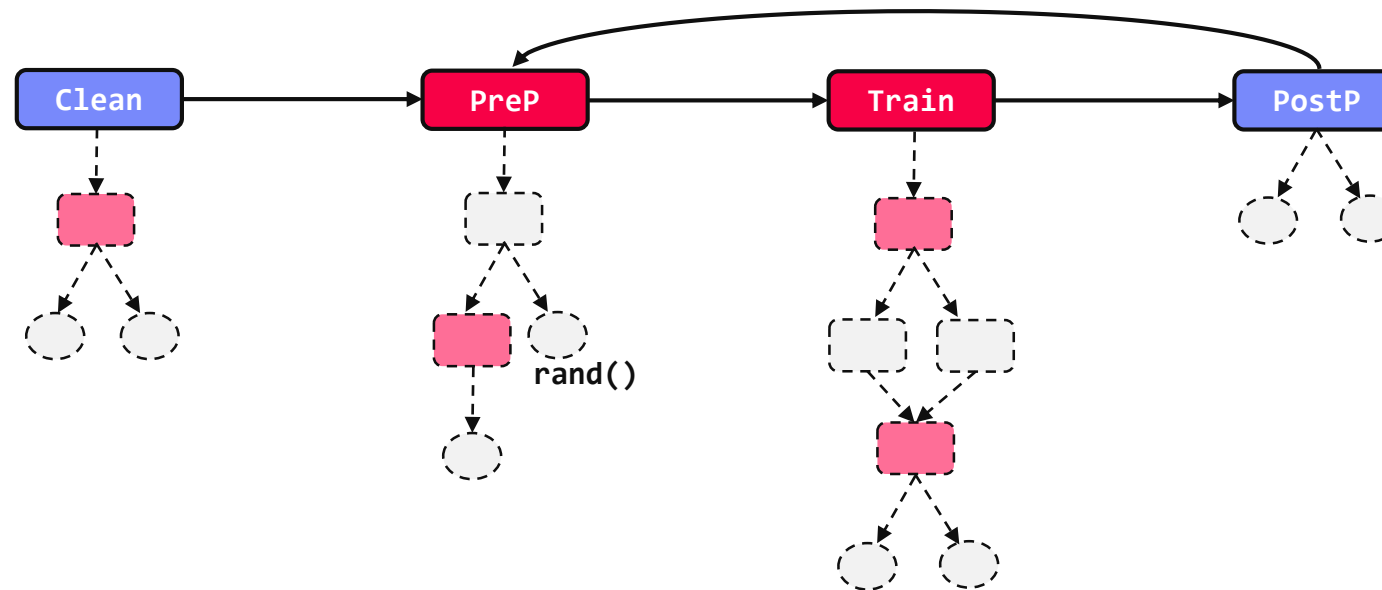
Coarse-grained Reuse

Existing Approaches

- **Coarse-grained** lineage tracing of top-level tasks
- **Black-box view** of individual steps (hidden substeps)
- Cannot eliminate **fine-grained redundancy**
- Fail to detect internal **non-determinism** (`rand()`, random reshuffling and initialization, drop-out layers)

[Doris Xin et al: Helix: Holistic Optimization for Accelerating Iterative Machine Learning. **PVLDB 12, 4 (2018)**]

[Behrouz Derakhshan et al: Optimizing Machine Learning Workloads in Collaborative Environments. **SIGMOD 2020**]



Sources of Redundancy

Running Example: Grid Search Hyper-parameter Tuning for LM

User Script

```
X = read('data/X.csv')
y = read('data/y.csv')
for(i in 1:10) {
  s = sample(15, ncol(X));
  [loss, B] = gridSearch('lm',
    'l2norm', ...,
    list('reg', 'icpt', 'tol')
  print(loss+ "for feature set s");
}
```

Internal Built-in Functions

```
lm = function(...) {
  if (ncol(X) <= 1024)
    B = lmDS(X,y,icpt,reg)
  else
    B = lmCG(X,y,icpt,
      reg,tol)
}
```

```
lmDS = function(...) {
  if (icpt > 0) {
    X = cbind(X, Ones);
    if (icpt == 2)
      X = scaleAndShift(X)
  } ...
  l = matrix(reg,ncol(X),1)
  A = t(X) %*% X + diag(l)
  b = t(X) %*% y
  beta = solve(A, b) ...}
```

```
lmCG = function(...) {
  if (icpt > 0) {
    X = cbind(X, Ones);
    if (icpt == 2)
      X = scaleAndShift(X)
  } ...
  while (i<maxi & nr2>tgt) {
    q = (t(X) %*% (X%*%ssX_p))
    p = -r + (nr2/old_nr2)*p;
  }
```

- #1 Redundant **lmDS** calls for tuning 'tol'
- #2 $X^T X$ and $X^T y$ in **lmDS** are independent of 'reg'
- #3 Same pre-processing block for **lmDS** and **lmCG**
- #4 Same **cbind** calls for 'icpt' = 1 and 2
- #5 Partially overlapping $X^T X$ and $X^T y$ with **cbind** of Ones
- #6 Random feature sets exhibit overlapping features

Redundancy across data science lifecycle tasks

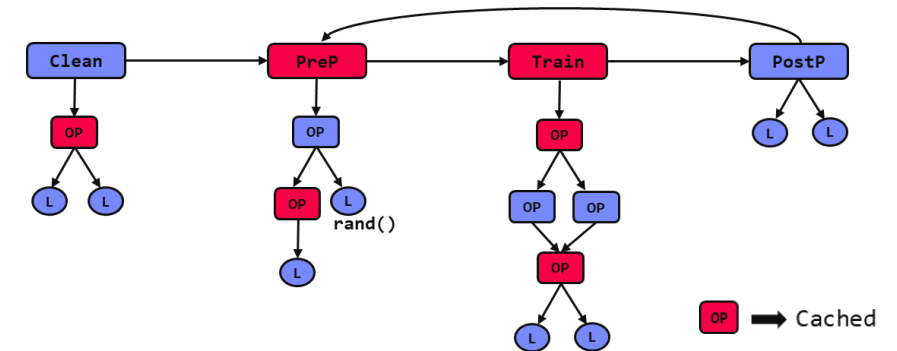
Introducing LIMA

■ Lineage/Provenance as Key Enabling Technique

- Model versioning, **reuse of intermediates**, incremental maintenance, auto differentiation, and **debugging** (results and intermediates, convergence behavior via query processing over lineage traces)

■ LIMA

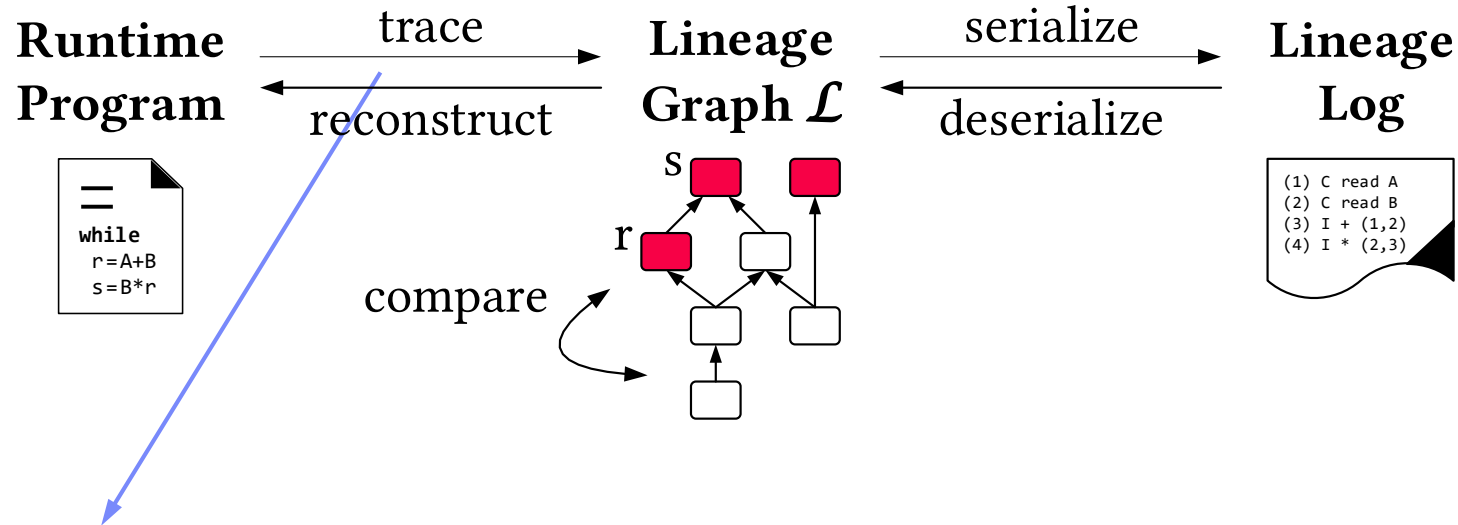
- A framework for **fine-grained** lineage tracing and reuse **inside ML systems**
- Efficient, **low-overhead** lineage tracing of individual operations
- **Full and partial reuse** across the program hierarchy



Lineage Tracing

(Key Operations and Lineage Deduplication)

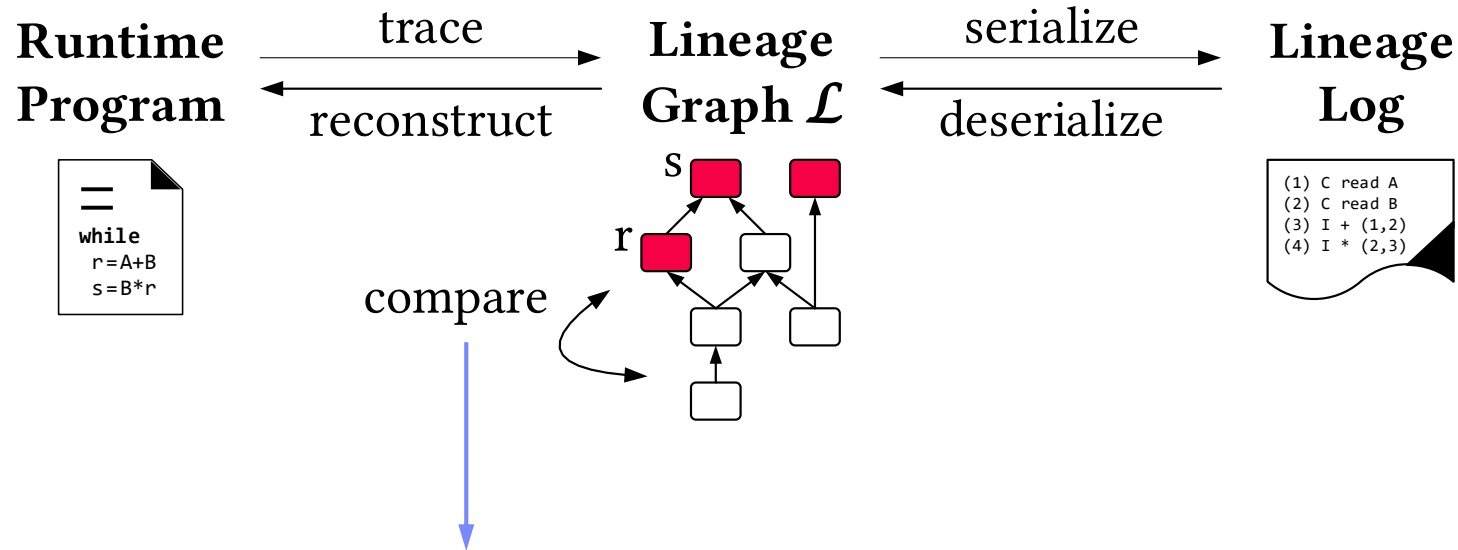
Basic Lineage Tracing



Lineage Tracing
Lifecycle

- a) **Efficient Lineage Tracing**
 - Trace lineage of logical operations for **all live variables**
 - Tracing of inputs, literals, and **non-determinism**
 - **Immutable** lineage DAG
 - Execution context maintains LineageMap that **maps live variable names to lineage items**

Basic Lineage Tracing

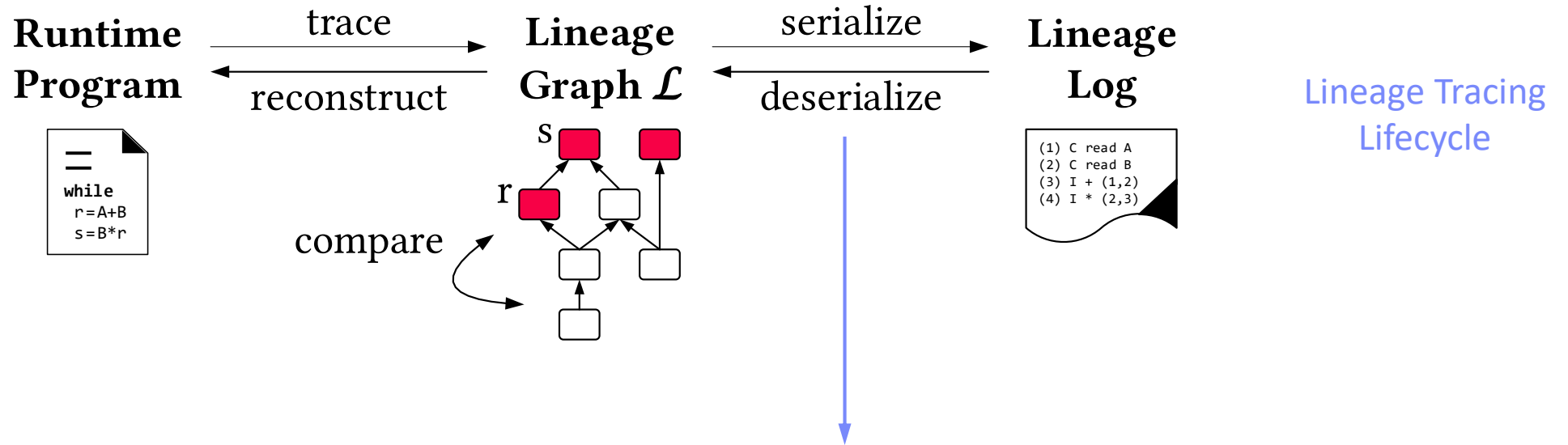


Lineage Tracing
Lifecycle

■ b) Comparison of Lineage DAGs

- Lineage items implement `hashCode()` and `equals()`
- Hash over the hashes of opcode, data item, and all inputs
- **Non-recursive** equals; returns true if the opcode, data and all inputs are equivalent

Basic Lineage Tracing

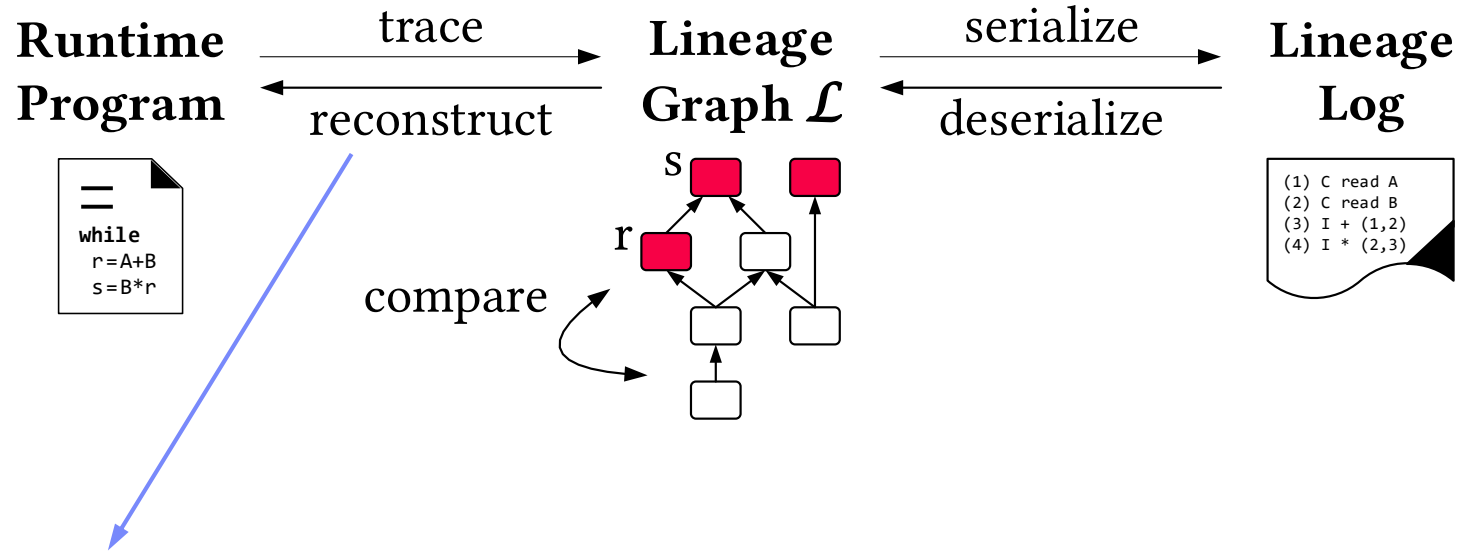


Lineage Tracing
Lifecycle

■ c) Serialization and Deserialization of Lineage DAGs

- `lineage(x)`, and `write(x, 'f')` always generates '`f.lineage`'
- Serialization unrolls the DAG in a depth-first manner
- Deserialization converts lineage log into a lineage DAG

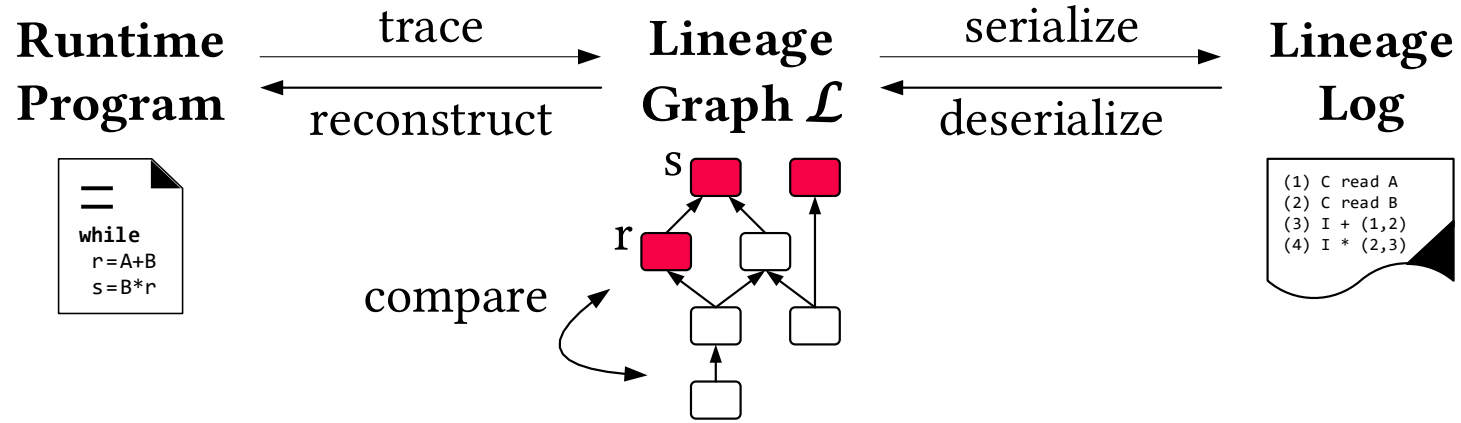
Basic Lineage Tracing



Lineage Tracing
Lifecycle

- **d) Re-computation from Lineage**
 - Generate runtime program from a lineage DAG
 - Compute exactly the same intermediates
 - Does not contain control flow
 - $X = \text{eval}(\text{deserialize}(\text{serialize}(\text{lineage}(X))))$

Basic Lineage Tracing



Lineage Tracing
Lifecycle

The entire lifecycle of lineage tracing with the key operations is very valuable as it **simplifies testing, debugging and reproducibility**.

Lineage Deduplication

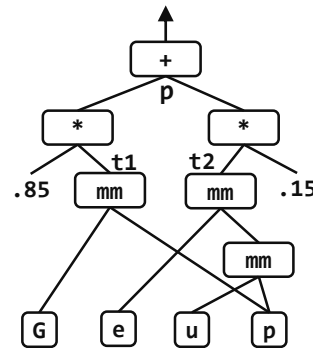
Problem

- Very **large lineage DAGs** for mini-batch training (repeated execution of loop bodies)
- NN training w/ 200 epochs, batch-size 32, 10M rows, 1K instructions \rightarrow 4TB \rightarrow **4GB w/ deduplication**

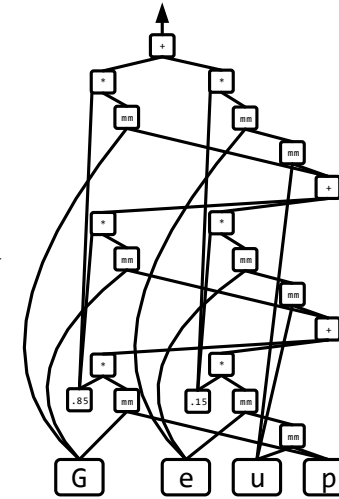
```
for(i in 1:3) {
  t1 = G %% p;
  t2 = e %% (u %% p);
  p = .85*t1 + .15*t2;
}
```

PageRank

single
iteration

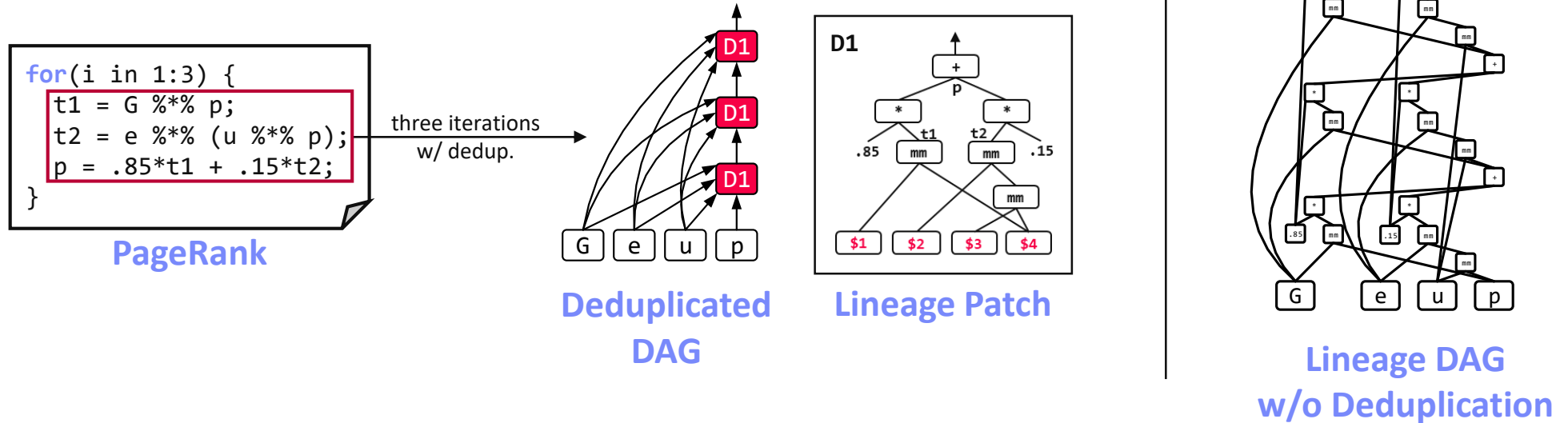


three
iterations



Lineage DAG

Lineage Deduplication



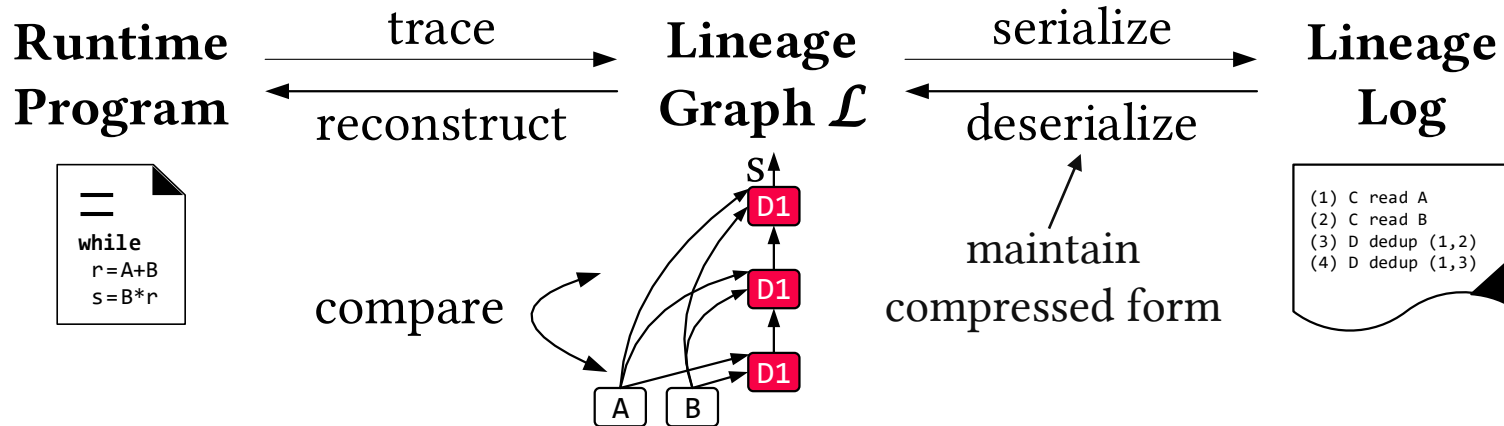
■ Solution

- Trace **each independent path once**; store as patches
- Refer to the patches via **single lineage items**

■ Implementation

- Proactive setup: **count** distinct control paths
- Runtime of iterations: trace lineage, **track taken path**
- Post-iteration: save the patch, add a **single dedup lineage item** to the global DAG

Operations on Deduplicated Graphs



Integration

- Last-level **for**, **parfor**, **while** loops and functions
- **Non-determinism**: add seeds (e.g. dropout layers) as input placeholders
- **Compare** regular and deduplicated DAGs
- Serialize, deserialize, re-compute **w/o causing expansion**

Lineage-based Reuse

(Lineage Cache, Multi-level Reuse, Partial Reuse and Eviction Policies)

Lineage-based Reuse

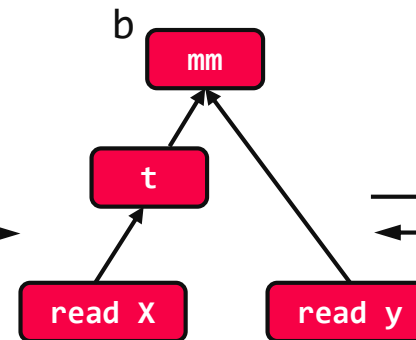
Operation-Level Full Reuse

- Lineage cache comprises a hash map, **Map<Lineage, Intermediate>**
- Before executing instruction, **probe lineage cache** for outputs
- Leverage compare functionality via efficient `hashCode()` and `equals()`

```
lmDS = function(...) {
  [...]
  A = t(X) %*% X + diag(1)
  b = t(X) %*% y
  beta = solve(A, b) ...}
```

Closed-Form Linear Regression

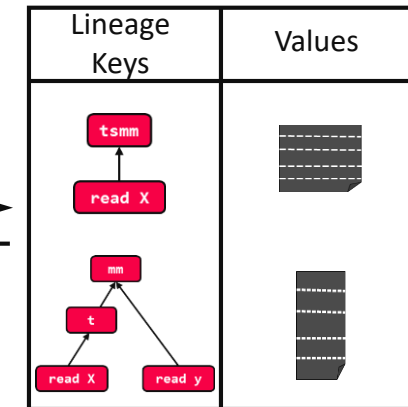
trace



Lineage DAG for $X^T Y$

probe

reuse

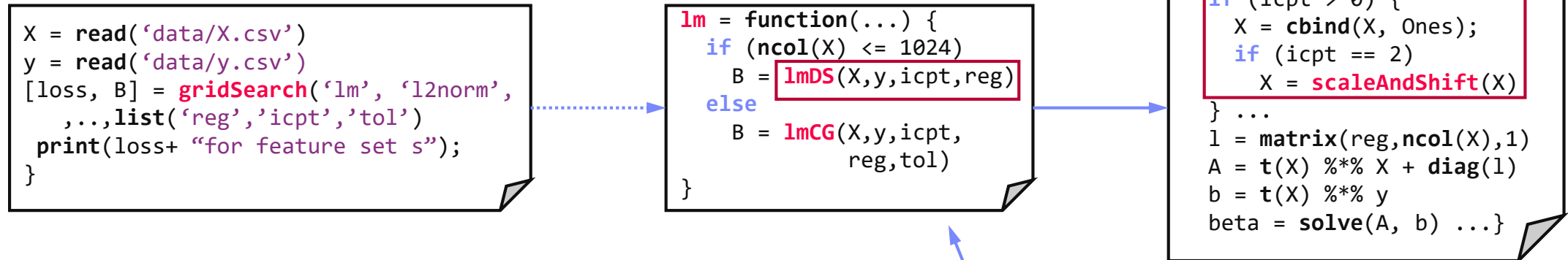


Lineage Cache

Multi-Level Full Reuse

Limitations of Operation-level Reuse

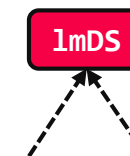
- Fails to remove **coarse-grained redundancy**, e.g., entire function
- Cache pollution and interpretation overhead



Solution: Multi-level Reuse

- **Hierarchical** program structure as reuse points
- Mark if deterministic during compilation
- **Special lineage item** to represent a function call
- Avoid cache pollution and interpretation overhead
- Similar to function, reuse **code blocks**

→ Redundant lmDS calls
→ Redundant preprocessing block for icpt = 1, 2



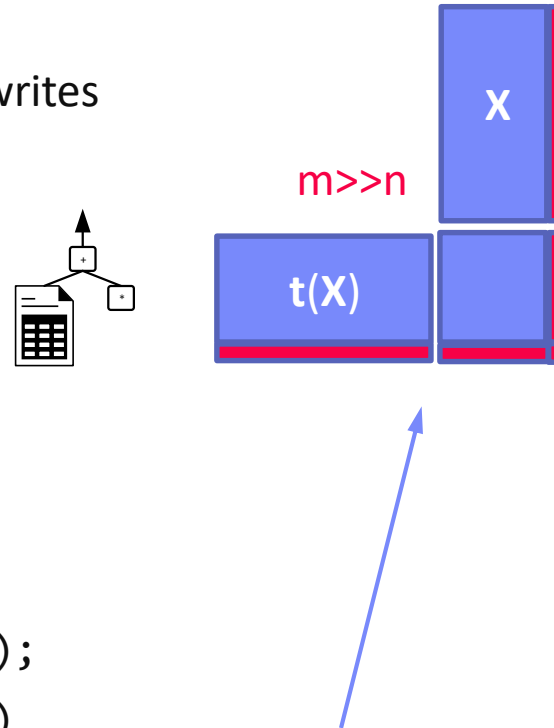
Partial Operation Reuse

Limitations of Full Reuse

- Often partial results overlap. Example: stepLM

Solution: Partial Reuse

- Reuse partial results via dedicated rewrites (compensation plans)
- Probe ordered-list of rewrites of source-target patterns
- Construct **compensation plan**, compile and execute
- Based on real use cases



```
stepLM = function(...) {
  while (continue) {
    parfor (i in 1:n) {
      if (!fixed[1,i]) {
        Xi = cbind(Xg, X[,i])
        B[,i] = lm(Xi, y, ...)
      }
    }
    # add best to Xg
    # (AIC)
  }
}
```

```
lmDS = function(...) {
  l = matrix(reg, ncol(X), 1)
  A = t(X) %*% X + diag(l)
  b = t(X) %*% y
  beta = solve(A, b) ...}
```

$O(n^2(mn^2+n^3)) \rightarrow O(n^2(mn+n^3))$

Example Rewrites

- #1 $\text{rbind}(X, \Delta X)Y \rightarrow \text{rbind}(XY, \Delta XY);$
- #2 $X\text{cbind}(Y, \Delta Y) \rightarrow \text{cbind}(XY, X\Delta Y)$
- #3 $\text{dsyrk}(\text{cbind}(X, \Delta X)) \rightarrow \text{rbind}(\text{cbind}(\text{dsyrk}(X), X^T \Delta X), \text{cbind}(\Delta X^T X, \text{dsyrk}(\Delta X)))$
where, $\text{dsyrk}(X) = X^T X$
- ...

Cache Eviction

- **Delete or spill.** Spill to disk if re-computation time > estimated I/O time

Statistics and Cost

- Static: operation execution time, distance from leaves, in-memory and in-disk sizes
- Dynamic: last access timestamp, #accesses
- Estimate: disk I/O

Eviction Policies

- Determine **order of eviction**
- **LRU**: orders by normalized last access time
 - Pipelines with temporal reuse locality
- **DAG-Height**: orders by depth of DAG (deep lineage traces have less reuse potential)
 - Mini-batch scenario. Reuse across epochs
- **Cost&Size**: orders by cost, size ratio (preserve objects with high cost to size ratio) scaled by #accesses
 - Global reuse utility. Performs well in a wide variety of scenarios

Eviction Policies & Eviction Orders

Policy	Orders Objects by
LRU	normalized last access timestamp
Dag-Height	height of operation-DAG, descending
Cost&Size	cost/size * #accesses

[Behrouz Derakhshan et al: Optimizing Machine Learning Workloads in Collaborative Environments. **SIGMOD 2020**]



Default Policy

Integration with ML Systems

#1 Task-parallel Loops

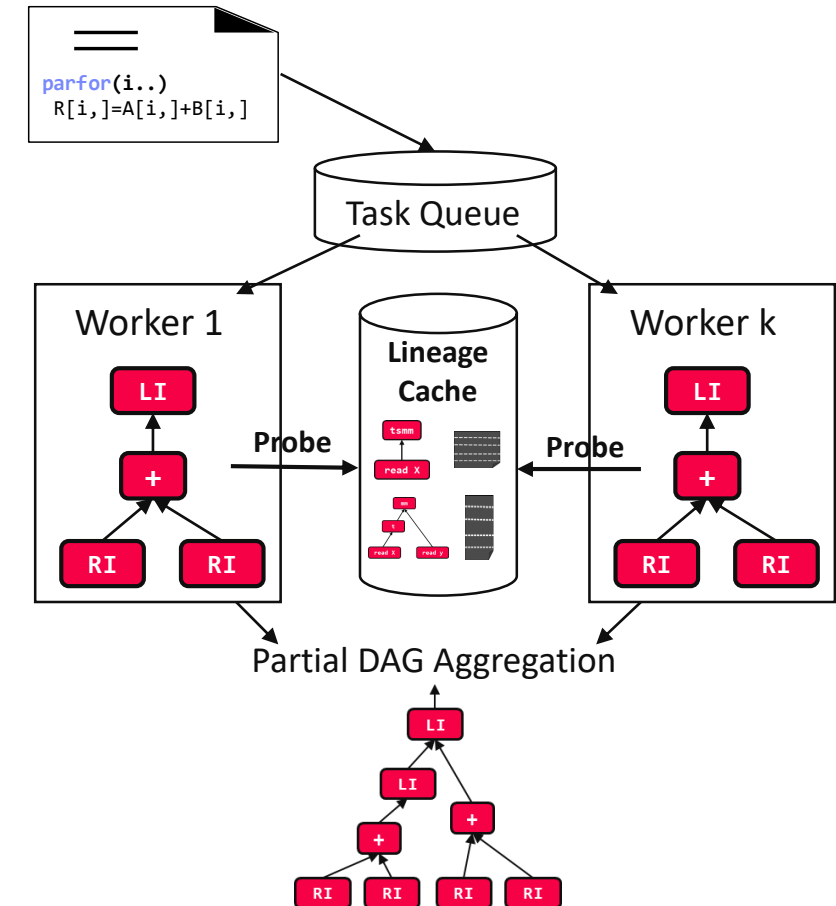
- **Worker-local** tracing. Merge in a linearized manner
- Tasks share lineage cache in a **thread-safe** manner
- Lineage cache “placeholders” to avoid redundant computation in parallel tasks

#2 Operator Fusion

- Fusion loses operator semantics
- Construct lineage patches **during compilation**, and add those to the global DAG during runtime

#3 Compiler Assistance

- **Unmark** not reusable operations for caching to avoid cache pollution and probing
- **Reuse-aware rewrites** during compilation to create additional reuse opportunities
- Reuse-aware rewrites during runtime **recompilation**



Lineage Tracing and Task Parallelism

Experiments

(End-to-end ML Pipelines, ML Systems Comparison)

Experimental Setting

■ Baselines



■ Datasets

Dataset	nrow(X_0)	ncol(X_0)	nrow(X)	ncol(X)	ML Alg.
APS	60,000	170	70,000	170	2-Class
KDD 98	95,412	469	95,412	7,909	Reg.

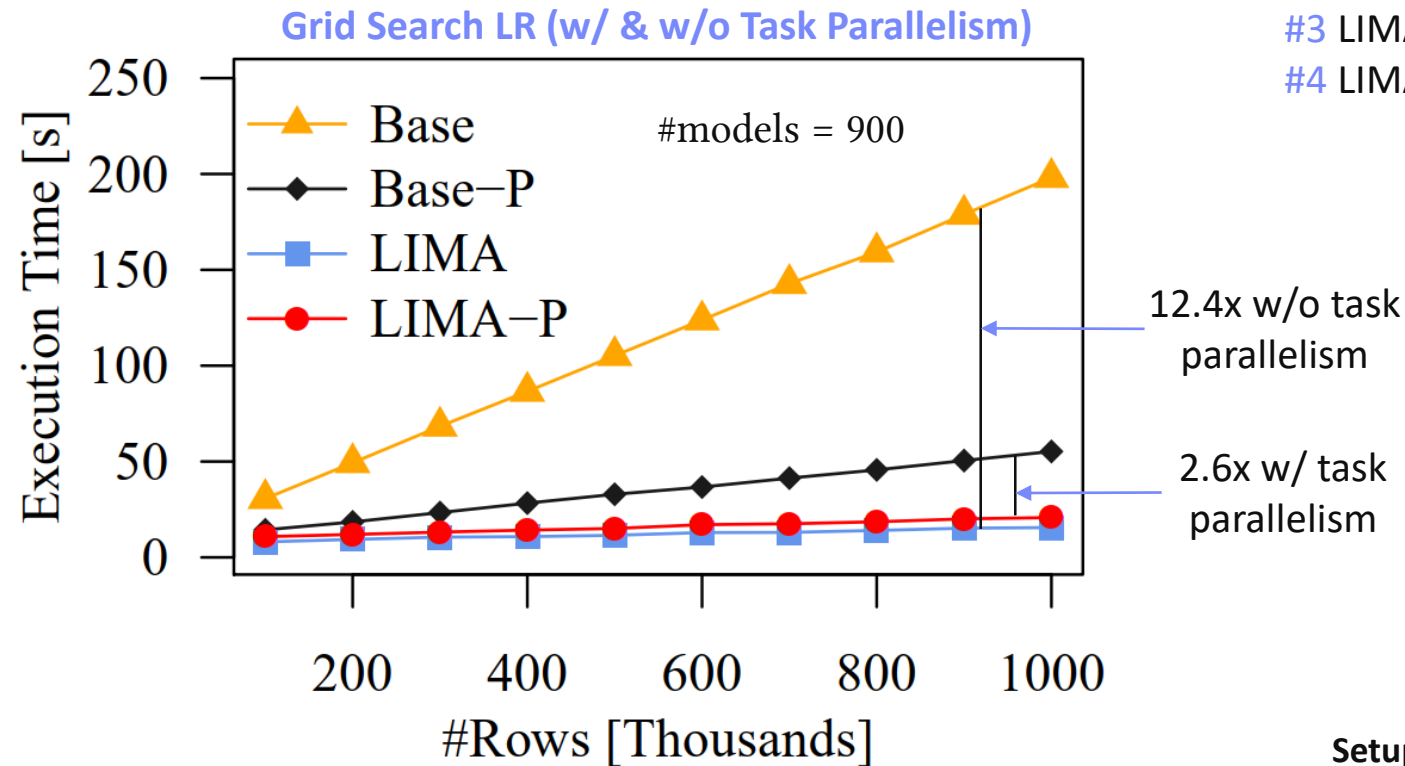
Lineage-based reuse is largely independent of data skew.

■ Workloads

Variety of end-to-end ML pipelines incl. data prep., feature engineering, traditional ML training (regression, classification) and NN training.

Experiments

■ End-to-end ML Pipelines



Baselines:

- #1 Base = **SystemDS** default config.
- #2 Base-P = Base **w/ task parallel** execution
- #3 LIMA = Lineage tracing and reuse
- #4 LIMA-P = LIMA **w/ task parallel** execution

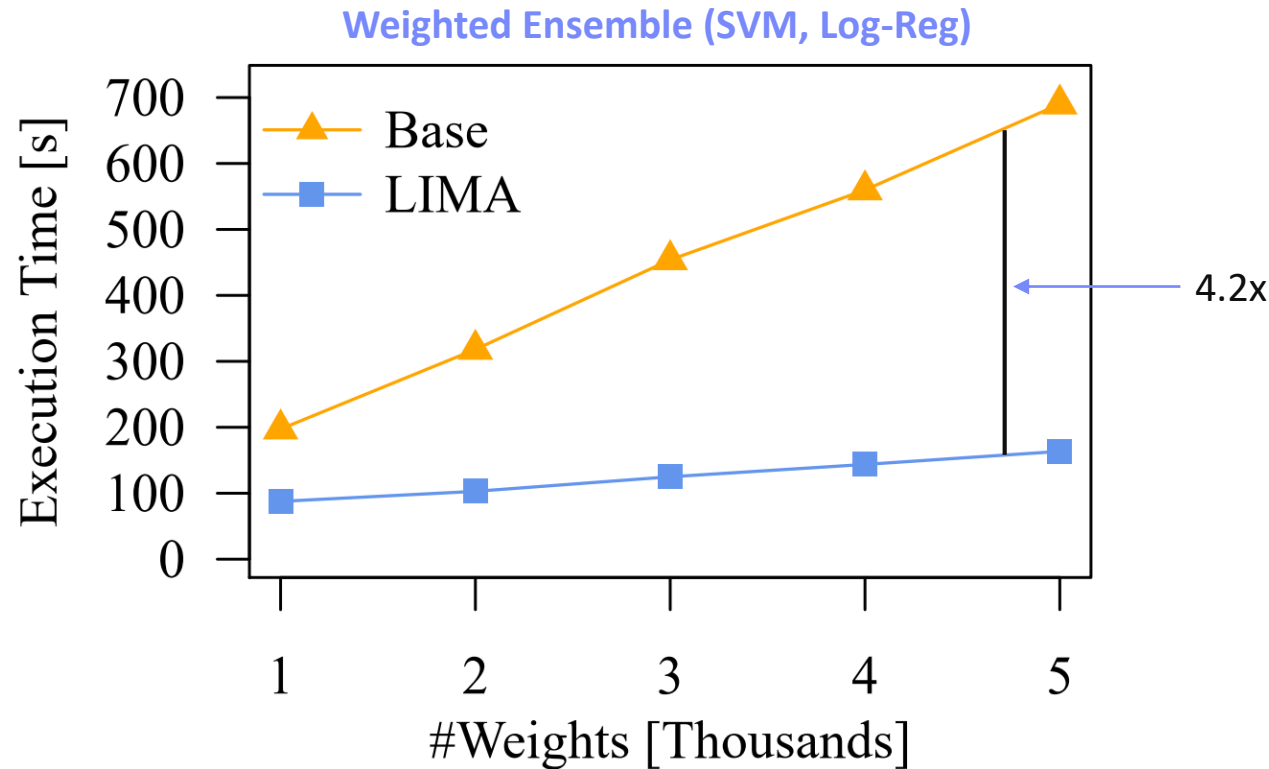
Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

Experiments

■ End-to-end ML Pipelines

Baselines:

- #1 Base = **SystemDS** default config.
- #2 LIMA = Lineage tracing and reuse

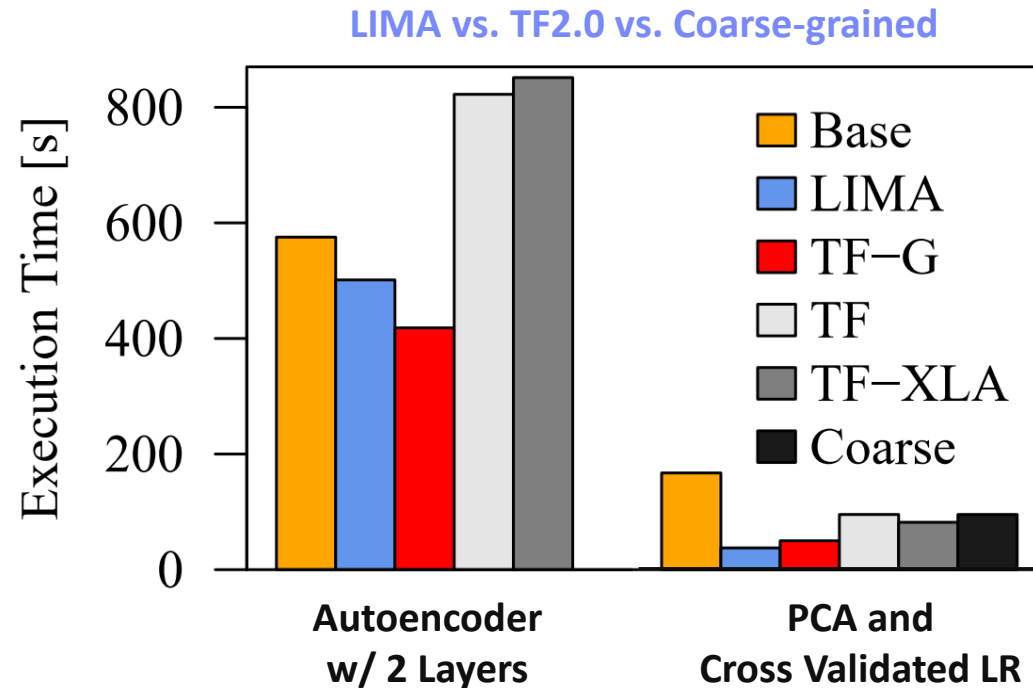


Large improvements due to **fine-grained redundancy elimination**

Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

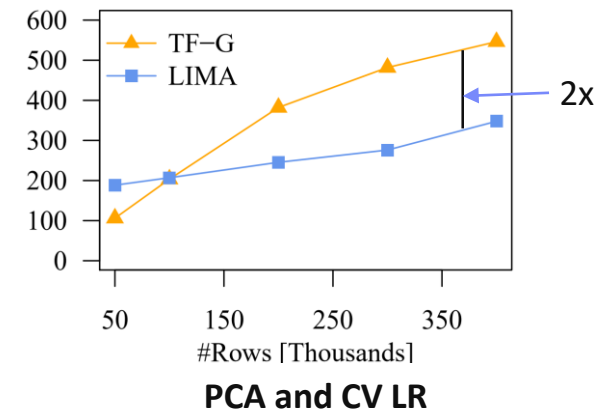
Experiments, cont.

ML Systems Comparison



Baselines:

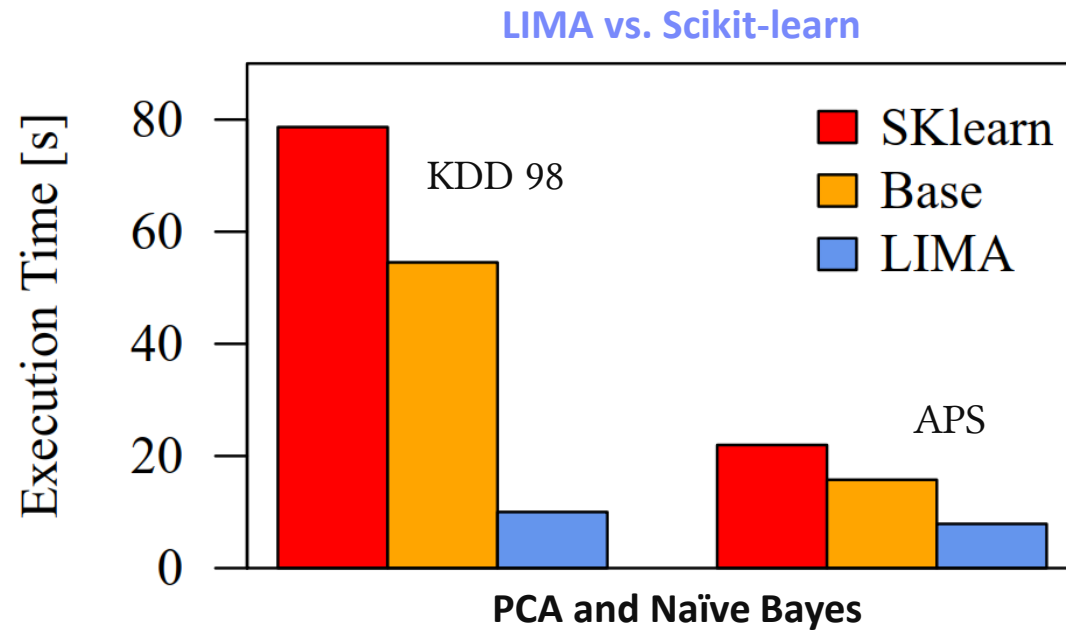
- #1 Base = **SystemDS** default config.
- #2 LIMA = Lineage tracing and reuse
- #3 TF-G = TensorFlow 2.3 **Graph mode**
- #4 TF = TensorFlow 2.3 Eager mode
- #5 TF-XLA = TF w/ XLA code gen. for CPU
- #6 Coarse = **Coarse-grained** reuse (HELIX)



Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

Experiments, cont.

ML Systems Comparison



Baselines:

#1 SKlearn = Scikit-learn

#2 Base = **SystemDS** default config.

#3 LIMA = Lineage tracing and reuse

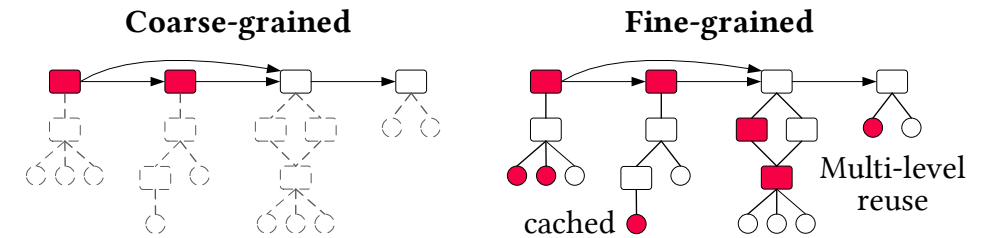
Competitive baseline performance against state-of-the-art ML Systems

Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

Conclusions

Summary

- **Fine-grained lineage tracing** in ML systems
- **Deduplication** for loops to reduce overhead
- Compiler-assisted **full, partial and multi-level reuse**
- Support for **fused operators and task-parallelism**



Conclusion

- Increasing redundancy is inevitable and difficult to address by library developers
- Compile time **CSE is only partially effective** due to conditional control flow
- Compiler-assisted runtime-based lineage cache proved effective

Future Work

- Combine with persistent materialization of intermediates
- Multi-location and multi-device caching
- Extend lineage support for model debugging and fairness constraints