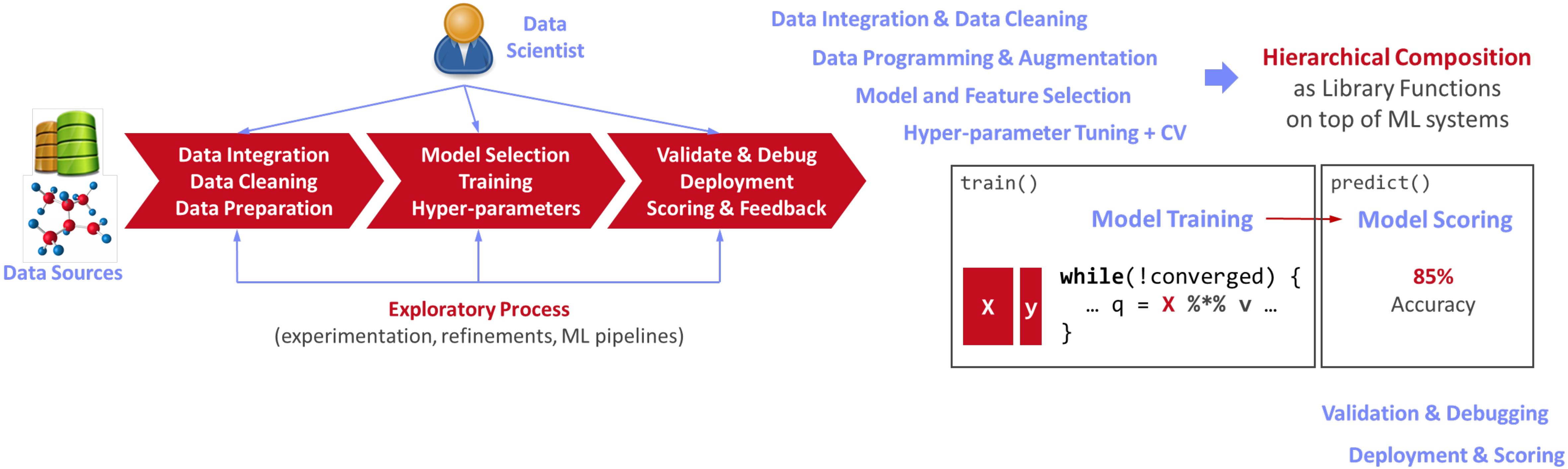
Fine-grained Lineage Tracing and Reuse in Multi-backend ML Systems

Arnab Phani
Technische Universität Berlin
Big Data Engineering (DAMS Lab)

Exploratory Data Science



- **Problem**
  - Data integration, cleaning and preparation techniques are themselves based on ML.
  - High **computational redundancy** in hierarchically composed ML pipelines
  - **Reproducibility** and **explainability** of trained models (data, parameters, prep)

Apache SystemDS [https://github.com/apache/systemds]

APIs: Command line, JMLC, Python
Spark MLContext, Spark ML,
(Scalable Algorithms + Primitives)

DML Scripts

Language

Compiler

Runtime

Write Once,
Run Anywhere

Apache SystemML™

07/2020 Renamed to **Apache SystemDS**
05/2017 Apache Top-Level Project
11/2015 Apache Incubator Project
08/2015 Open Source Release

[SIGMOD'15,'17,'19,'21abc,'23abc]
[PVLDB'14,'16ab,'18,'22]
[ICDE'11,'12,'15]
[CIDR'17,'20]
[VLDBJ'18]
[CIKM'22]
[DEBull'14]
[PPoPP'15]

In-Memory Single Node
(scale-up)

Hadoop or Spark Cluster
(scale-out)

Federated
(LA progs, PS)

In-Progress:

GPU

since 2014/16

since 2012

since 2010/11

since 2015

since 2019

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) ...}
```

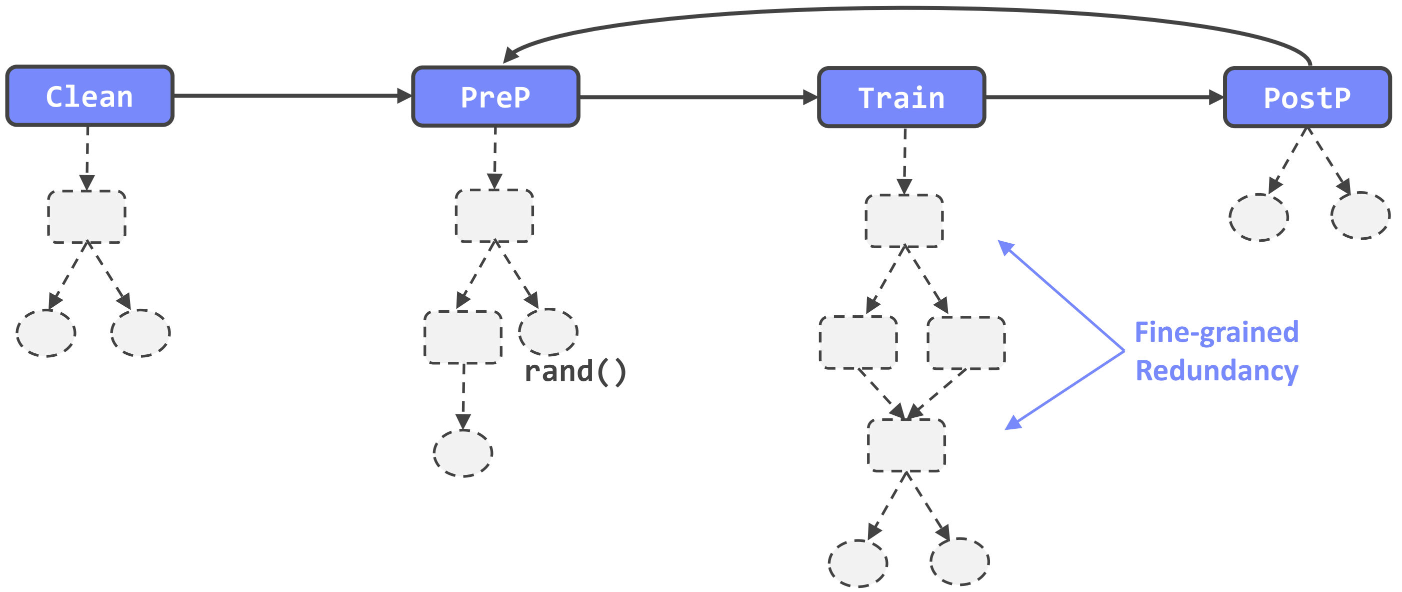
```
lmCG = function(...) {
  if (icpt > 0) {
    X = cbind(X, Ones);
    if (icpt == 2)
      X = scaleAndShift(X);
  } ...
  while (i<maxi & nr2>tgt) {
    q = (t(X) %*% (X%*%ssX_p))
    p = -r + (nr2/old_nr2)*p;
}
```

- #1 Redundant `lmDS` calls for tuning `'tol'`
- #2 $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$ 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 $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$ with `cbind` of Ones
- #6 Random feature sets exhibit overlapping features

Redundancy across data science lifecycle tasks

Coarse-grained Reuse

- **Existing Approaches**
  - **Coarse-grained** lineage tracing of top-level tasks
  - Compile time **CSE** and hand-optimization
- **Problem**
  - **Black-box view** of individual steps (hidden substeps)
  - Cannot eliminate **fine-grained redundancy**
  - Fail to detect internal **non-determinism** (`rand`, `sample,..`)




```
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");
}
```

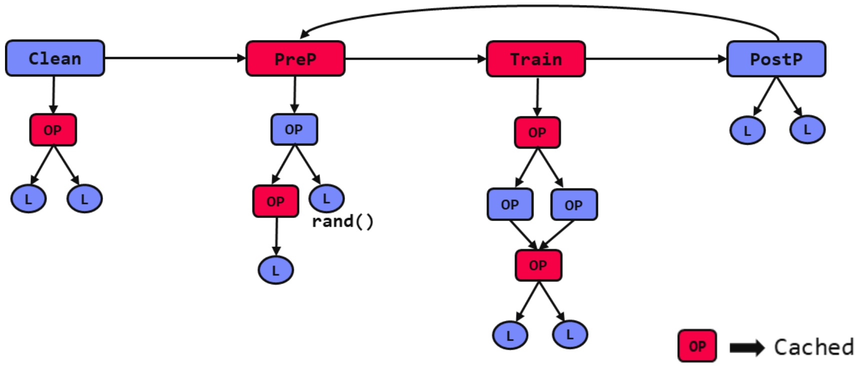
```
lm = function(...) {
  if (ncol(X) <= 1024)
    B = lmDS(X,y,icpt,reg)
  else
    B = lmCG(X,y,icpt,
             reg,tol)
}
```

```
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) ...}
```

Introducing LIMA [CIDR '20, SIGMOD '21]

- **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



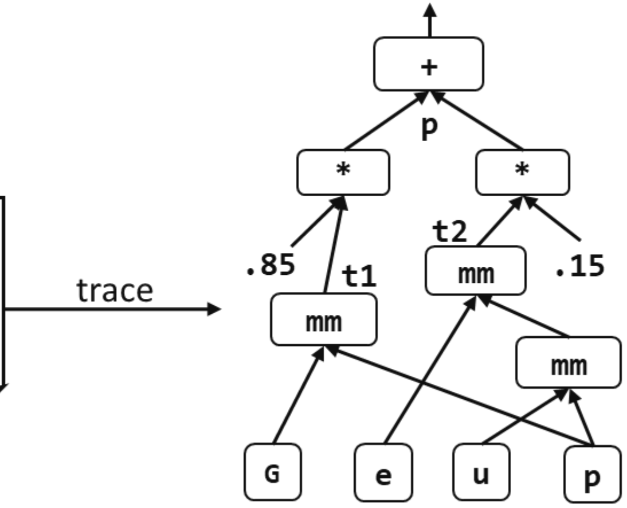
- **LIMA is integrated into Apache SystemDS**

Apache SystemDS ™

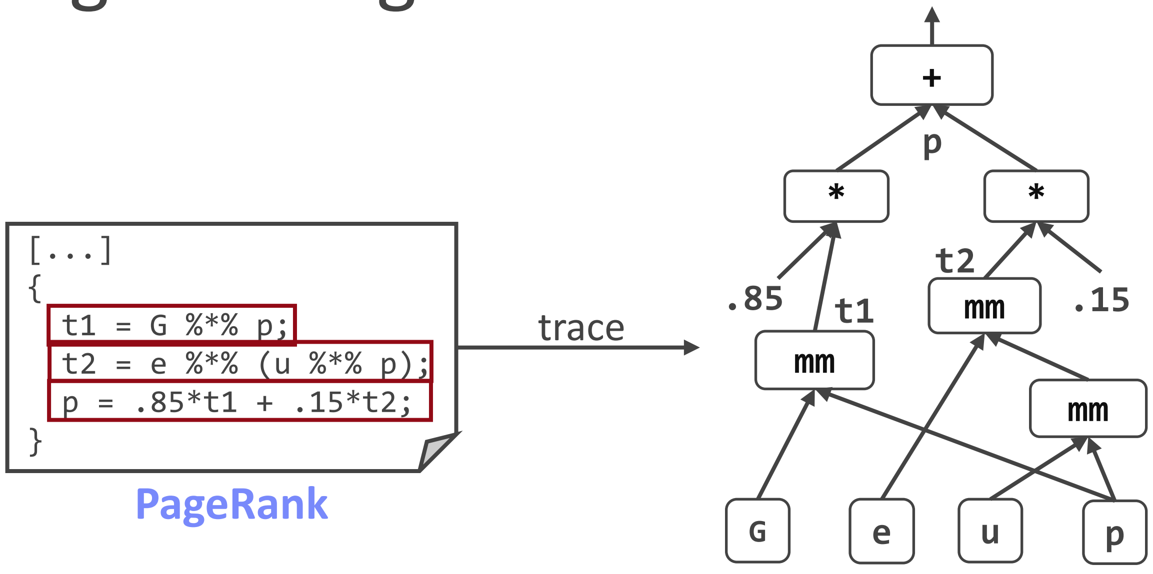
Lineage Tracing

(Key Operations and Lineage Deduplicaton)

```
[...]
{
  t1 = G %*% p;
  t2 = e %*% (u %*% p);
  p = .85*t1 + .15*t2;
}
```

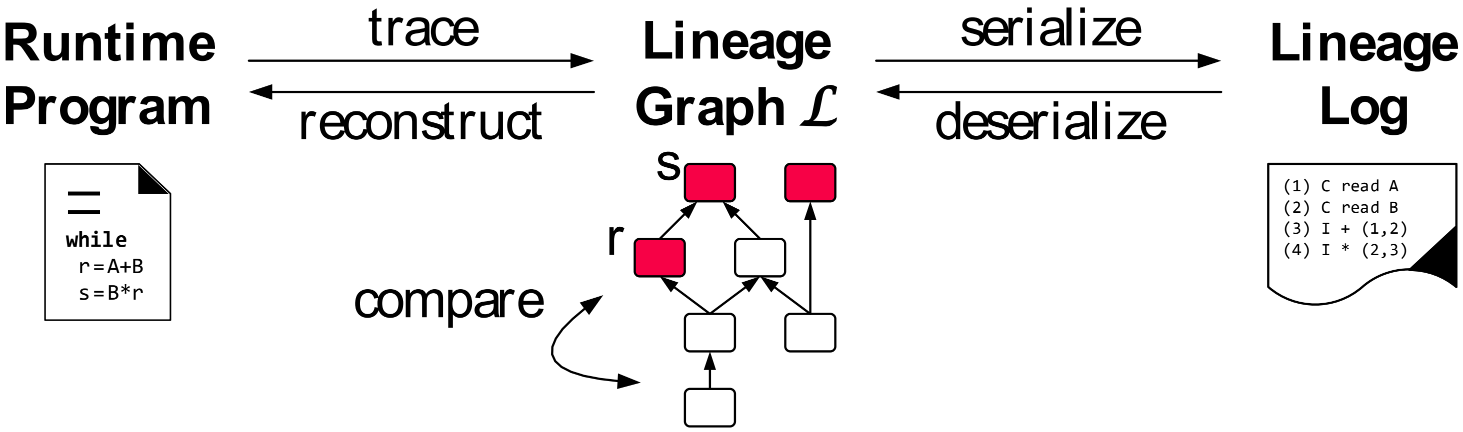


Basic Lineage Tracing

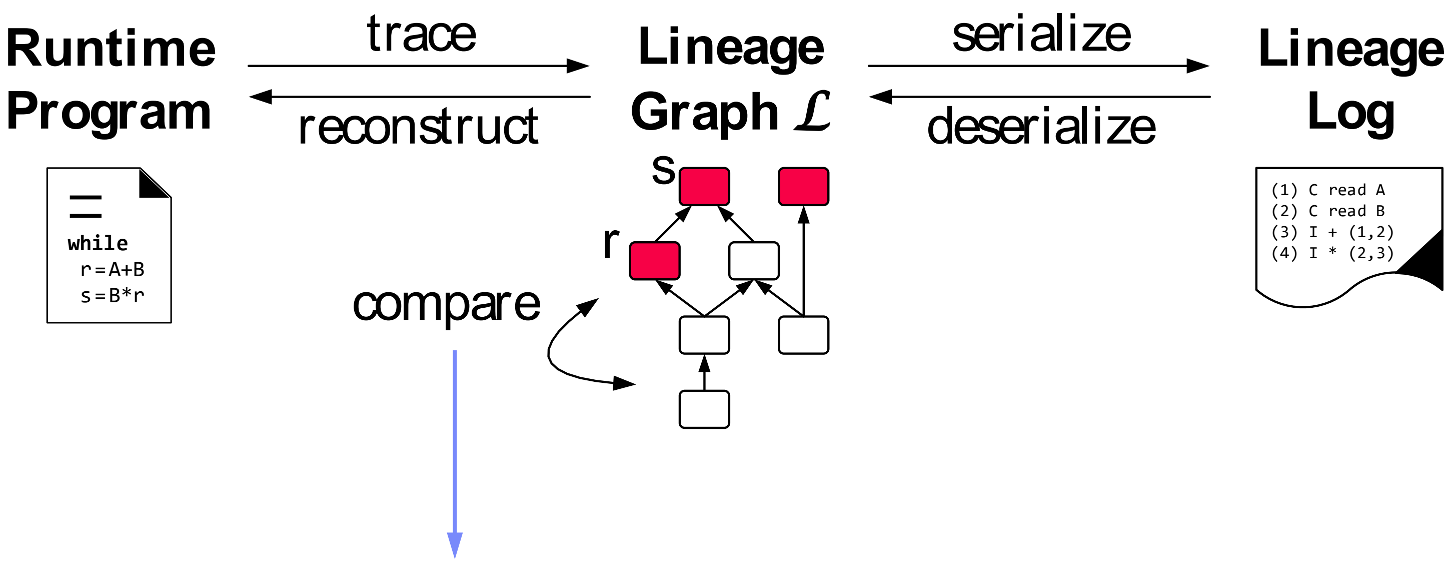


- Trace all live variables
- Trace non-determinism
- Incrementally built
- Immutable lineage DAG

Lineage Tracing Lifecycle



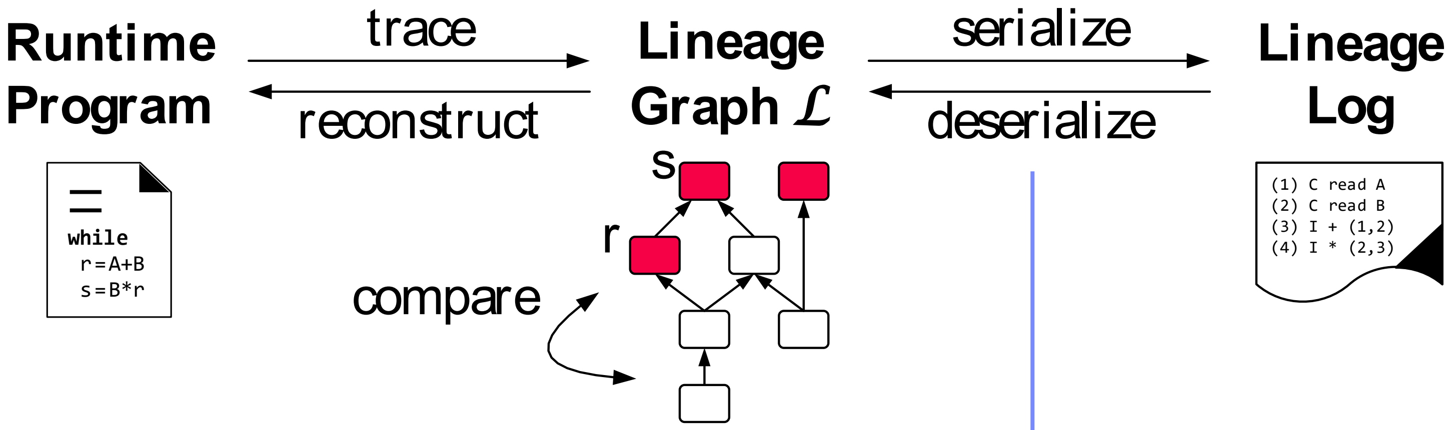
Basic Lineage Tracing



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

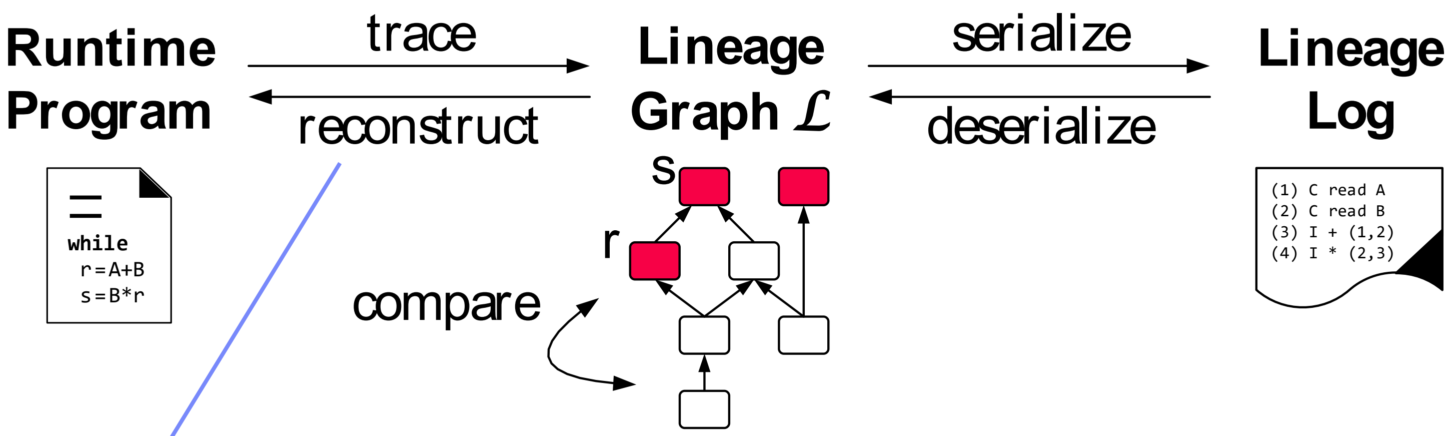
Basic Lineage Tracing



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

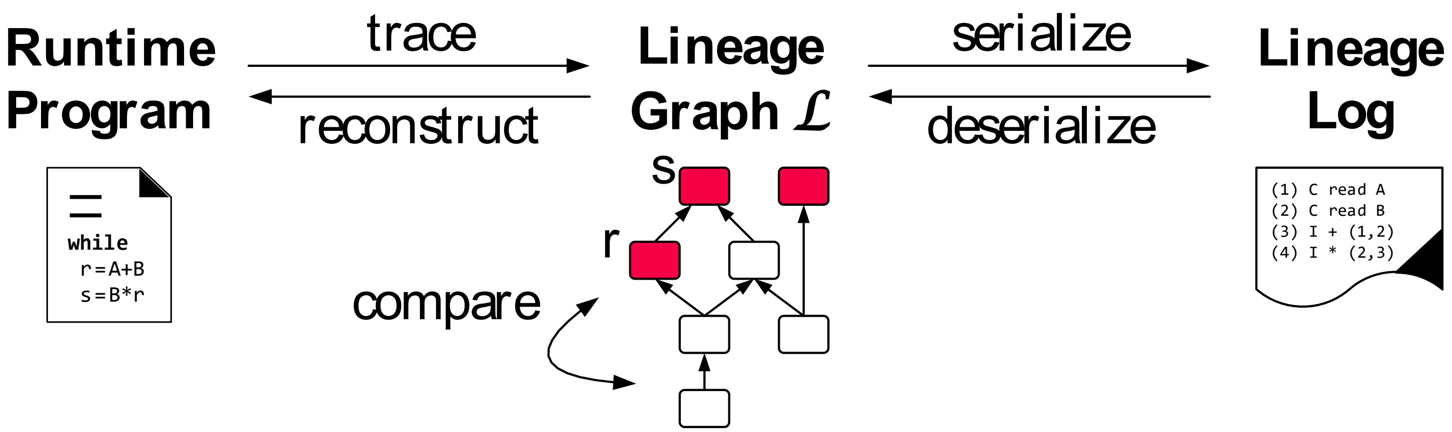
Basic Lineage Tracing



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 = eval(deserialize(serialize(lineage(X))))`

Basic Lineage Tracing

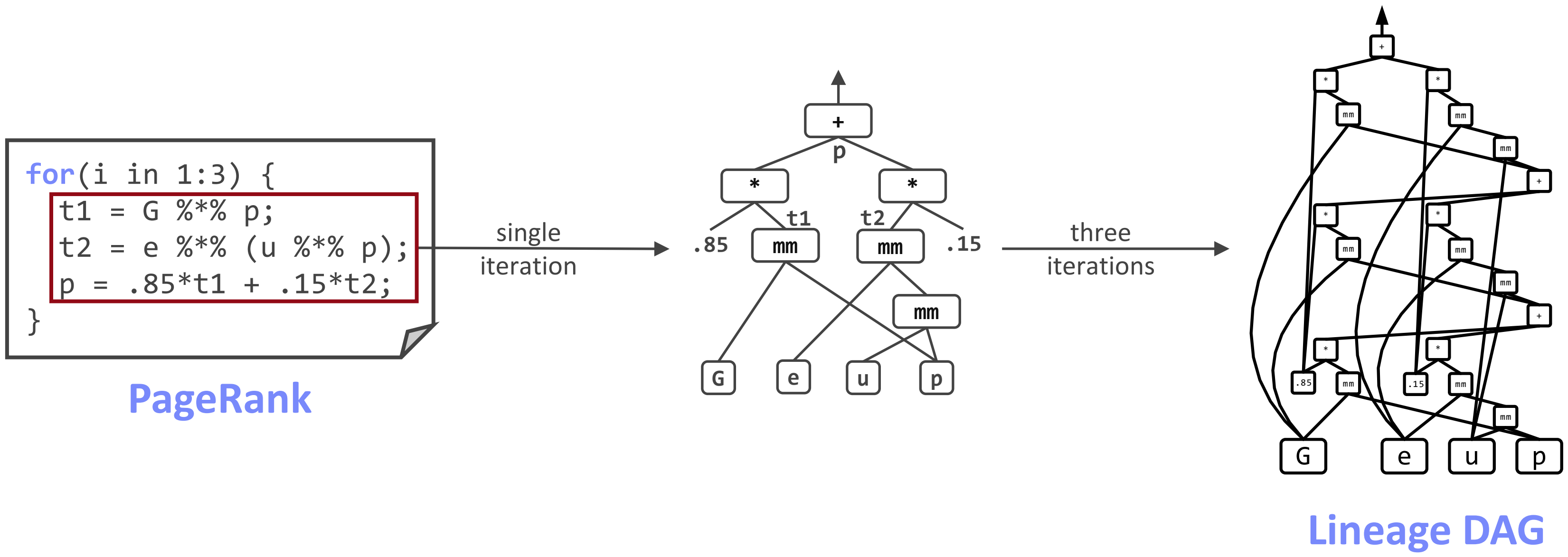


Lineage Tracing Lifecycle

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 → 4TB → **4GB w/ deduplication**



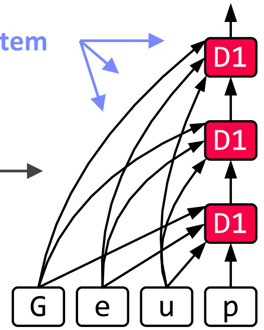
Lineage Deduplication

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

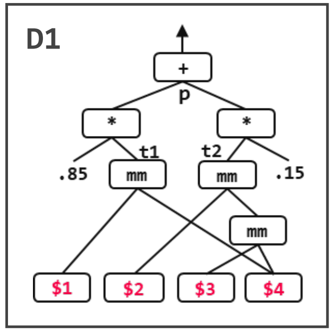
PageRank

three iterations
w/ dedup.

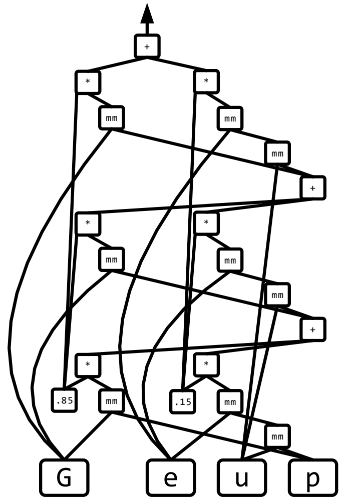
Single lineage item



Deduplicated DAG



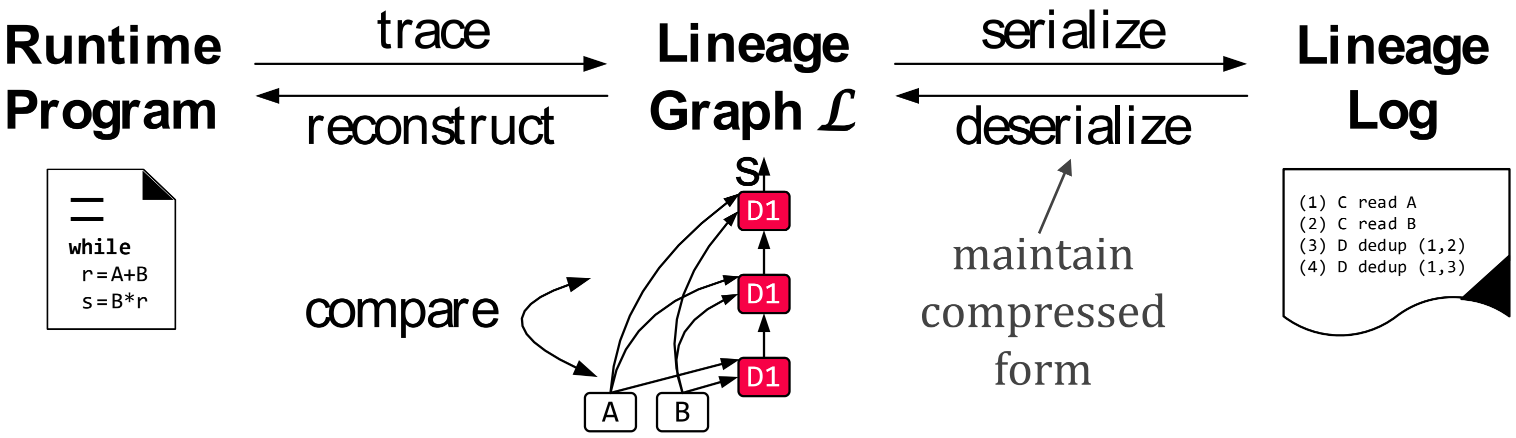
Lineage Patch



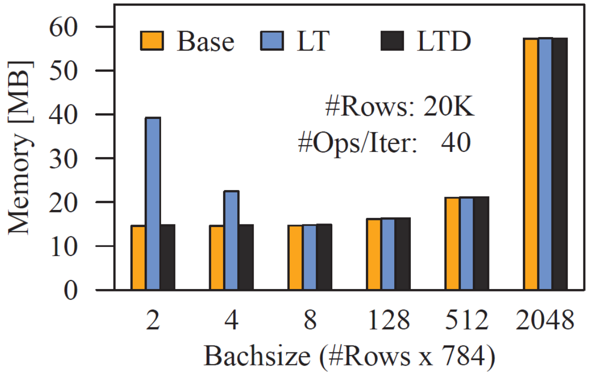
Lineage DAG
w/o Deduplication

- **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

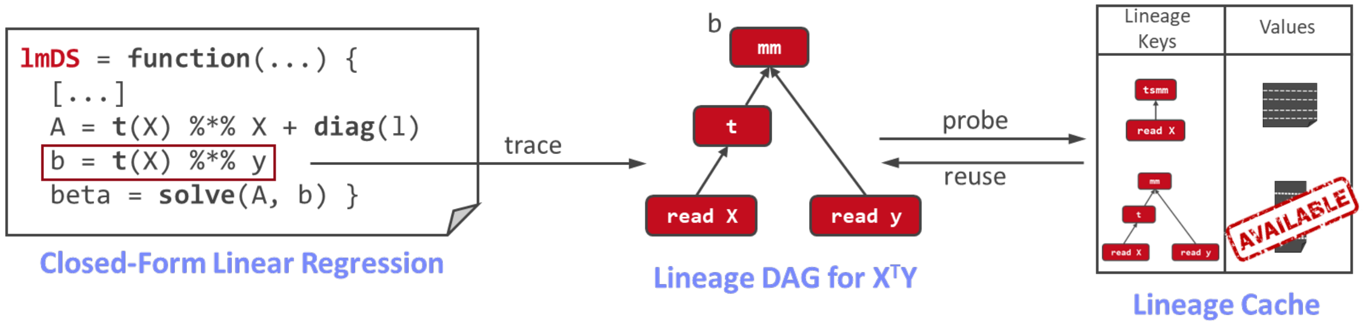


- **Integration**
  - Last-level `for`, `parfor`, `while` loops and functions
  - **Non-determinism**: add seeds as input placeholders
  - **Compare** regular and deduplicated DAGs
  - Serialize, deserialize, re-compute **w/o causing expansion**



Lineage-based Reuse

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

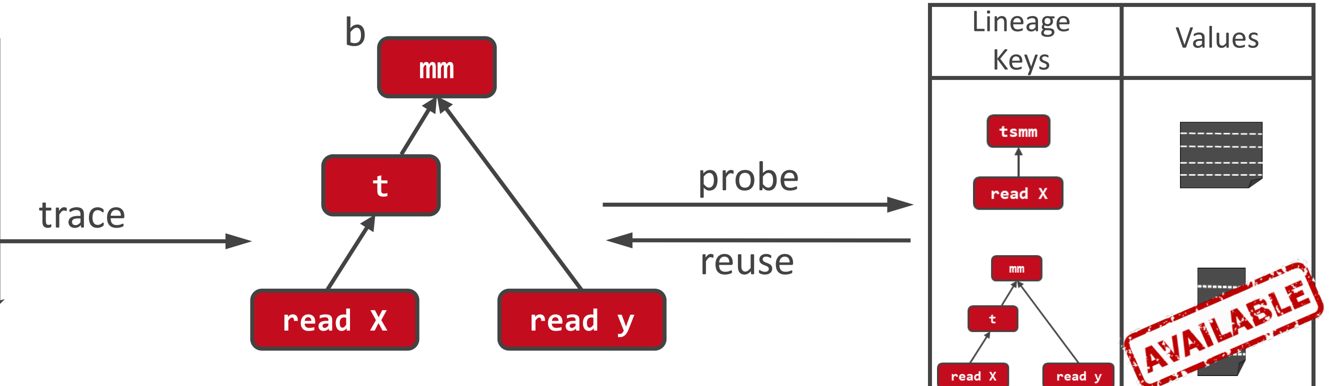


Lineage-based Reuse

- **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(l)
  b = t(X) %*% y
  beta = solve(A, b) }
```

Closed-Form Linear Regression



Lineage DAG for X^TY

Lineage Cache

- **Lineage Cache Eviction**
  - **Delete or spill**. Spill to disk if **re-computation time > estimated I/O time**
  - Policies: **LRU**, **DAG-Height**, **Cost&Size**
  - Policies determine **order of eviction**

Multi-Level Full Reuse

- **Limitations of Operation-level Reuse**
  - Fails to remove **coarse-grained redundancy**, e.g., entire function
  - Cache pollution and interpretation overhead

```
X = read(‘data/X.csv’)
y = read(‘data/y.csv’)
[loss, B] = gridSearch(‘lm’, ‘l2norm’,
   ,...,list(‘reg’,’icpt’,’tol’)
 print(loss+ “for feature set s”);
}
```

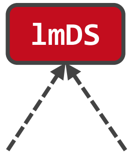
```
lm = function(...) {
  if (ncol(X) <= 1024)
    B = lmDS(X,y,icpt,reg)
  else
    B = lmCG(X,y,icpt,
             reg,tol)
}
```

```
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) ...}
```

→ **Redundant lmDS calls**

→ **Redundant preprocessing block for `icpt` = 1, 2**

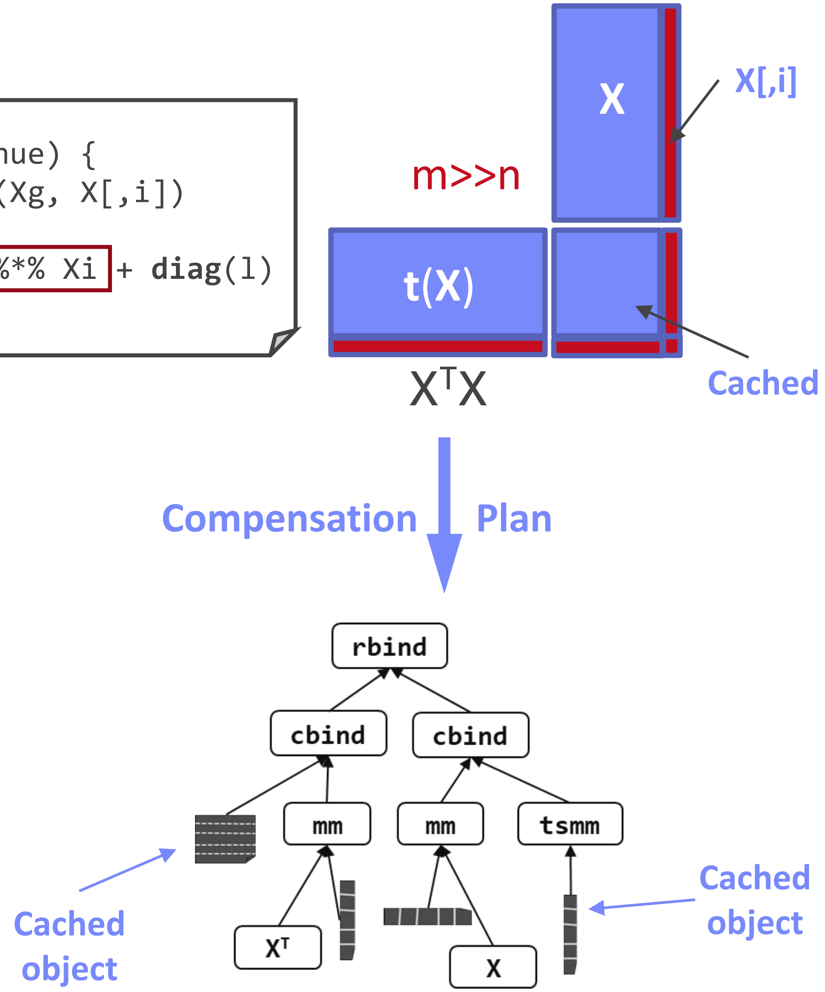
- **Solution: Multi-level Reuse**
  - **Hierarchical** program structure as reuse points
  - Mark if deterministic during compilation
  - **Special lineage item** to represent a function call
  - Avoid cache pollution and interpretation overhead
  - Similar to function, reuse **code blocks**



Partial Operation Reuse

- **Limitations of Full Reuse**
  - Often partial results overlap. Example: stepLM

```
while (continue) {
  Xi = cbind(Xg, X[,i])
  ...
  A = t(Xi) %*% Xi + diag(l)
  ... }
```



- **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

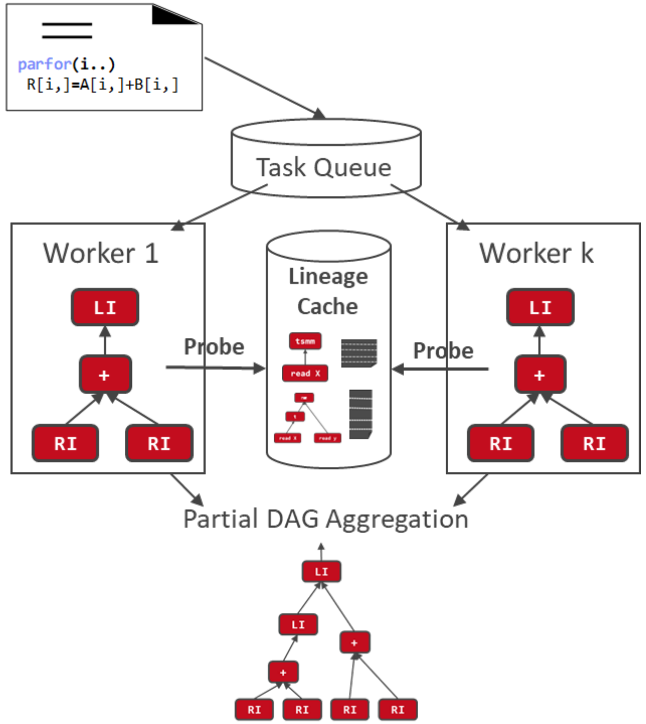
- **Example Rewrites**

  Cached

  - #1 **rbind**(X, ΔX)Y → **rbind**(XY, ΔXY);
  - #2 X**cbind**(Y, ΔY) → **cbind**(XY, XΔY)
  - #3 **dsyrk**(**cbind**(X, ΔX)) → **rbind**(**cbind**(**dsyrk**(X), $X^T\Delta X$), **cbind**($\Delta X^T X$, **dsyrk**(ΔX)))
  - ... where, dsyrk(X) = X^TX

Integration with Advanced Features

- **#1 Task-parallel Loops**
  - **Worker-local** tracing. Merge in a linearized manner
  - Reuse across tasks in a **thread-safe** manner
- **#2 Operator Fusion**
  - Construct lineage patches **during compilation**, and add those to the global DAG during runtime
- **#3 Compiler Assistance**
  - **Unmark not reusable operations** for caching
  - **Reuse-aware rewrites** during compilation to create additional reuse opportunities



Lineage Tracing and Task Parallelism

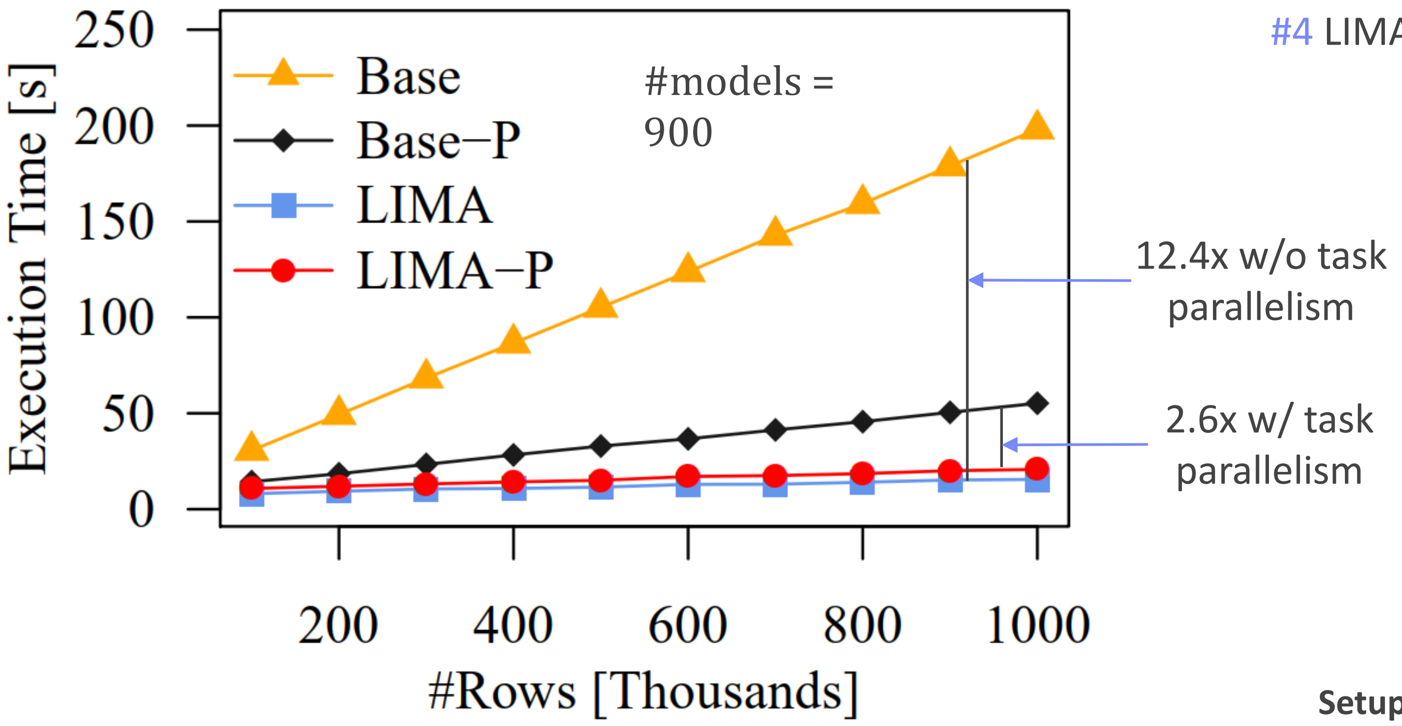
Experiments

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

Experiments

- **End-to-end ML Pipelines**

Grid Search LR (w/ & w/o Task Parallelism)



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

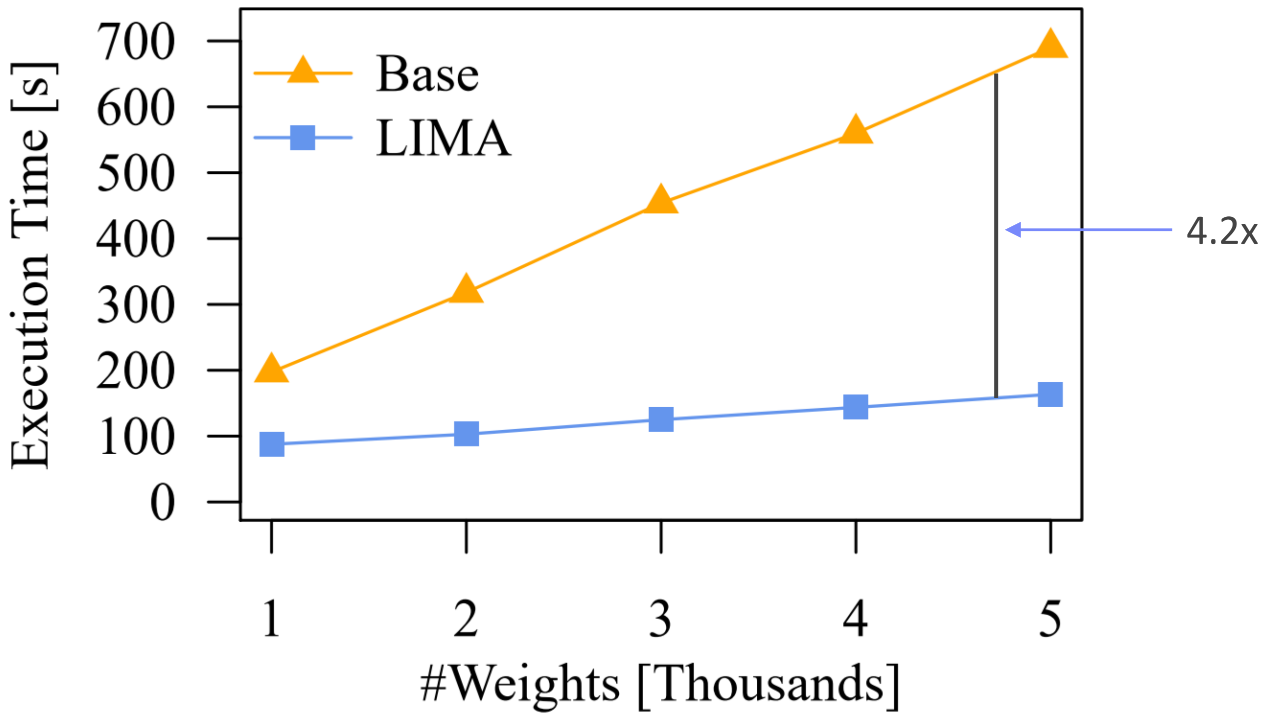
Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

Experiments

- **End-to-end ML Pipelines**

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

Weighted Ensemble (SVM, Log-Reg)



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

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

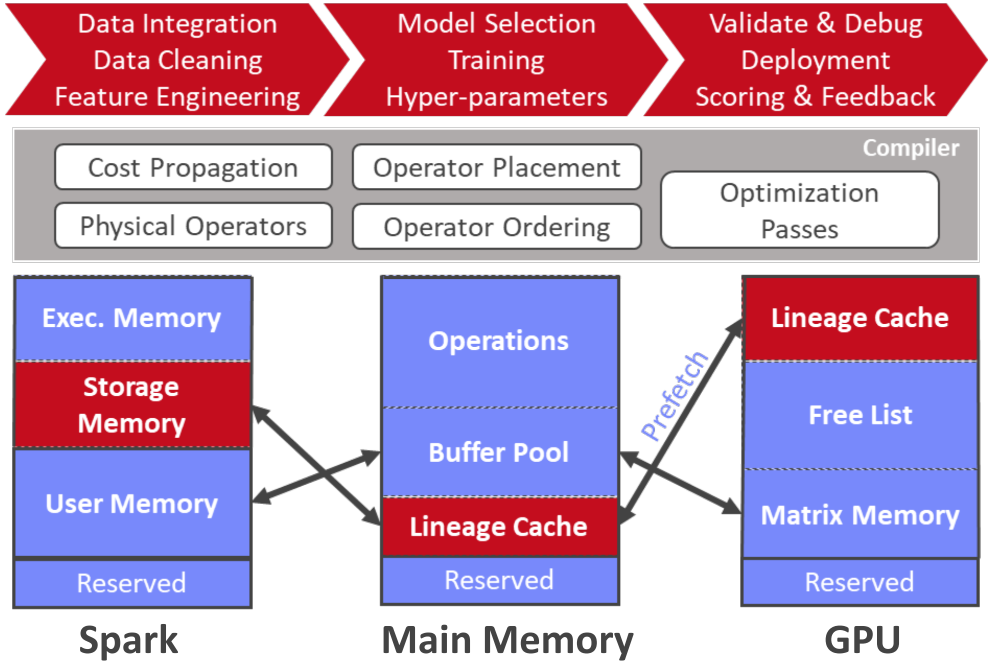
Deep Integration of Reuse in Multi-backend ML Systems

- **Memory Management**
  - **Varying sizes** and bandwidth (Spark backend, GPUs)
  - **Data exchange** across backends
  - Varying eviction policies of live intermediates
  - **Multi tenancy** with shared memory (Federated)

- **Execution Strategies**
  - **Lazy** (Spark), **asynchronous** (GPU),
  - Operator ordering and placement

- **Reuse and Lineage Cache**
  - Which intermediate to persist and where (storage level)
  - Multi-backend eviction policies for lineage cache
  - **Unified memory manager** including lineage cache

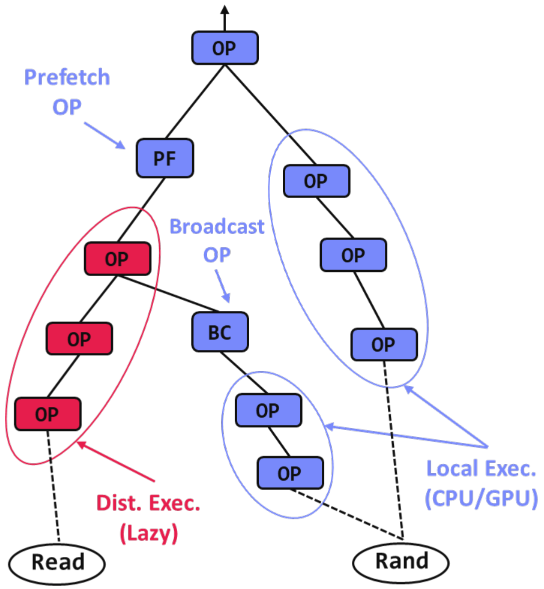
Memory management and reuse for complex ML pipelines



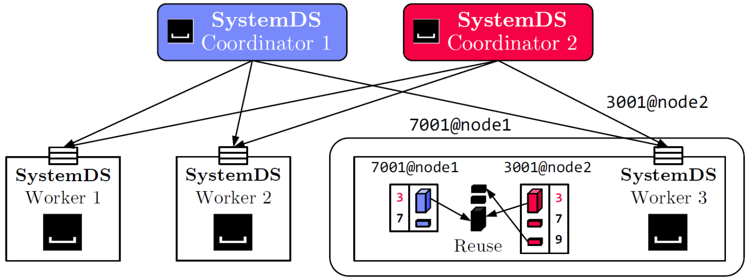
Apache SystemDS™ Spark DASK TensorFlow PYTORCH cuDNN cuBLAS

MUMBAI: Multi-backend Memory Management

[In Progress for PVLDB]



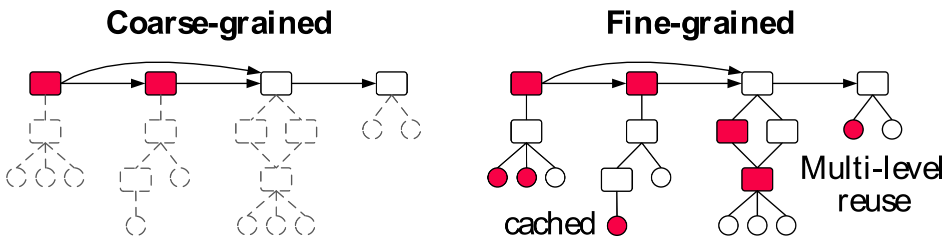
- **Asynchronous Operations**
  - Asynchronous data exchange: **prefetch**, early broadcast
  - Asynchronous triggering of lazy executions
- **Cost-based Operator Ordering**
  - **Minimize execution time** under memory constraints
  - Parallel executions, better resource utilization
- **Reuse in Spark**
  - Checkpoint placement for **remote caching** (speculative)
  - Cost-based, multi-level cache eviction
- **Reuse in GPU**
  - Unified memory manager
  - **Asynchronous cache eviction** to main memory
- **Reuse in Multi-tenant Federated Backend**
  - Shared cache, read sharing **[CIKM ’22 (Demo)]**
  - Lineage trace/CRS checksum exchange



Multi-tenant Federated Learning w/ Reuse

Conclusions

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



- **Conclusion**
  - Hierarchically composed complex ML pipeline contains increasing fine-grained redundancy
  - Difficult to address by library developers
  - Compile time **CSE is only partially effective** due to conditional control flow
  - Holistic integration of reuse relates to memory management, feature engineering

- **Future Work**
  - Combine with persistent materialization of intermediates
  - Extend lineage support for model debugging and fairness constraints