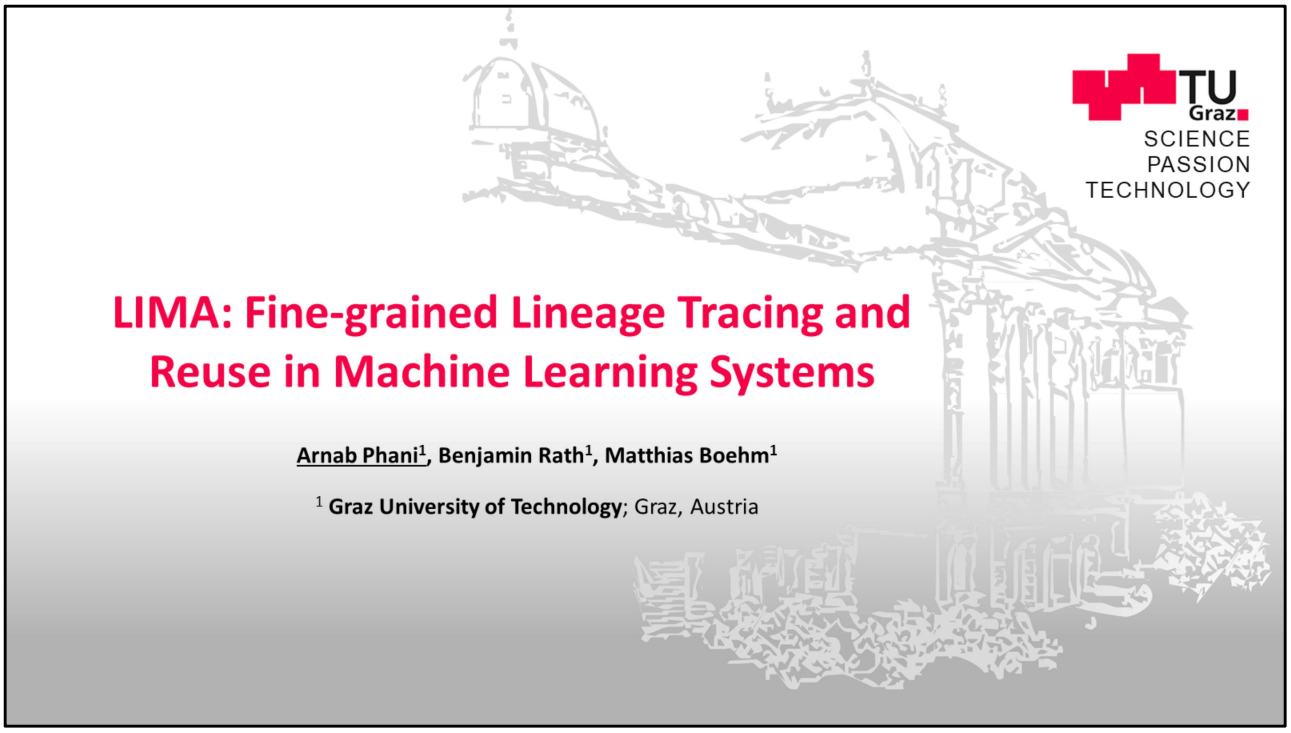
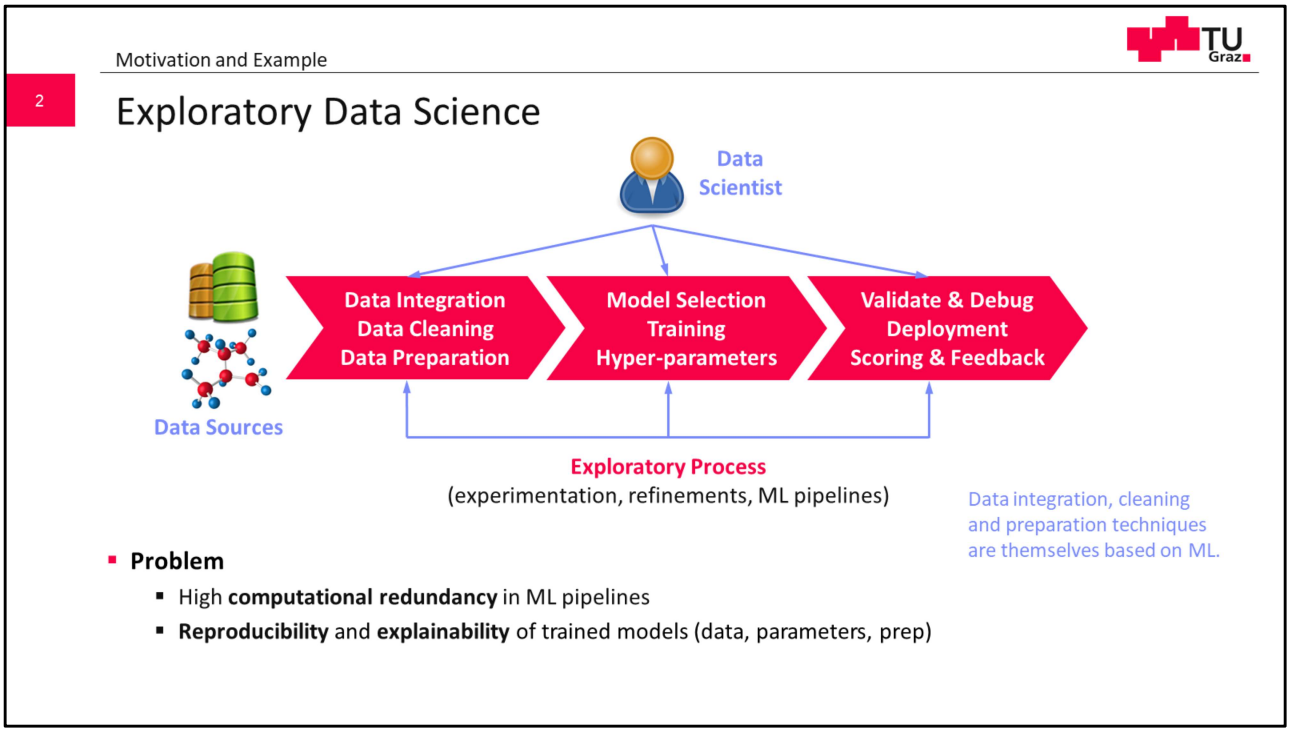


Hello all! My name is Arnab Phani. In this talk, I am going to present our paper LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems.



A typical data science workflow is inherently exploratory. Data scientists pose hypotheses, integrate the necessary data, and run ML pipelines of data cleaning, feature engineering, model selection and hyper-parameter tuning. Then the model gets validated and deployed for scoring. We also observed that the sate of the art data preparation techniques are themselves based on machine learning, which adds to the stack of exploratory ML tasks.

In this work, we point out 2 of the most important problems arise due to this repetitive nature of exploratory data science processes. First, increasingly complex, and hierarchically composed ML pipelines and workflows, create high computational redundancy. Second, explainability and reproducibility for the tasks become exceedingly difficult for the users.

Motivation and Example

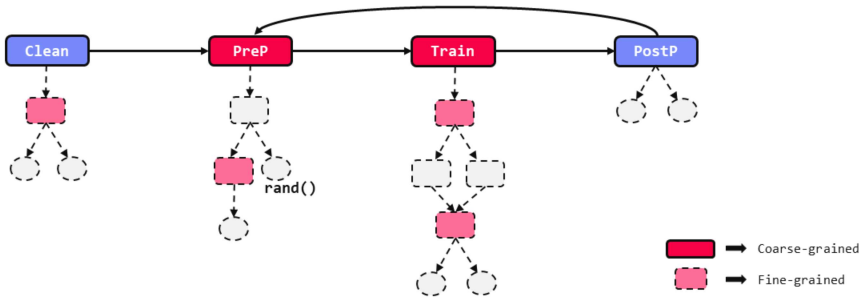
3

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)**]

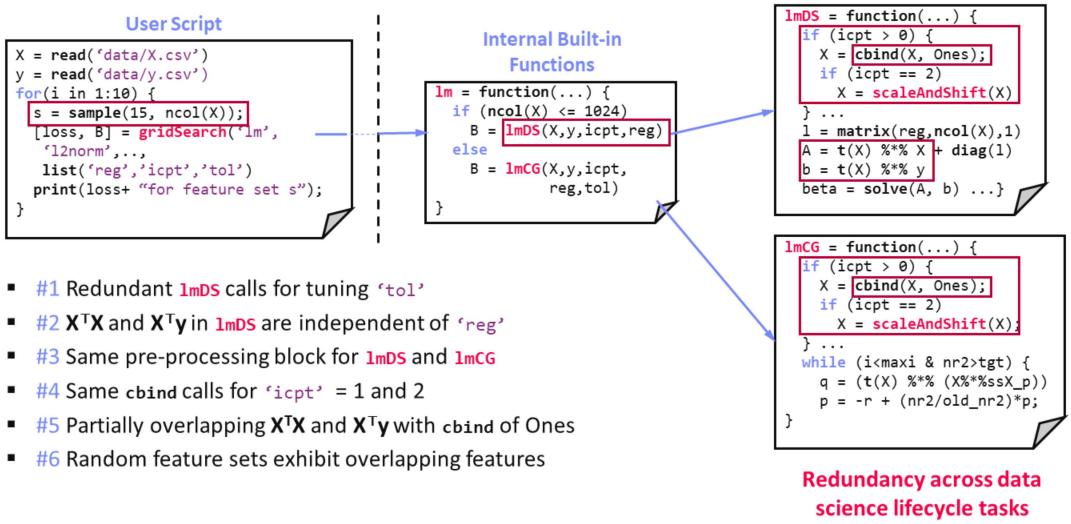
[Behrouz Derakhshan et al: Optimizing Machine Learning Workloads in Collaborative Environments. **SIGMOD 2020**]



Existing work, such as Helix and Collaborative Optimizer, addresses this high degree of redundancy with coarse-grained lineage tracing and reuse. Coarse-grained reuse substantially eliminate top-level redundancy over existing libraries. However, the top level ML methods are composed of hierarchy of functions and linear algebra operations. As shown in the figure, this approach views entire algorithms as black boxes, and thus, fails to eliminate fine-grained redundancy and fails to handle internal non-determinism. Non-determinism includes operations that yields different results for multiple runs, such random reshuffling and basic operations like random and sample.

Sources of Redundancy

- **Running Example: Grid Search Hyper-parameter Tuning for LM**



- #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

To understand the sources of these fine-grained redundancy better, I will use an running example of grid search hyperparameter tuning for linear regression. In this script, we read a feature matrix X and labels y, and extract 10 random subsets of 15 features. For each feature set, we tune 3 hyperparameters, regularization, intercept, and tolerance. Gridsearch function internally calls lm for each combination of hyperparameters. Based on the number of features, lm in turn dispatches, either to a closed-form method lmDS, or an iterative conjugate-gradient method, lmCG. Even this simple pipeline has many fine-grained redundancies and reuse opportunities.

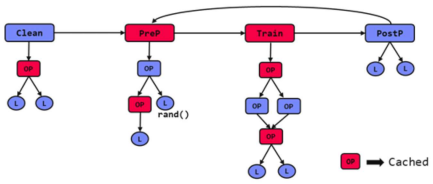
First, if X has fewer columns, all calls to lm are dispatched to lmDS and thus, hyperparameter tolerance is irrelevant. This leads to redundant calls of lmDS. Second, core operations of lmDS, X^TX and X^TY are independent of different regularization parameters, hence redundant. Third, pre-processing operations, including a call to `scaleAndShift`, are the same for lmDS and lmCG, and can be reused. Fourth, we perform the same `cbind` operation of X and a column vector of one, for all intercepts that are greater than 0. `Cbind` creates an intermediate larger than X, which makes it expensive. Fifth, re-executing X^TX and X^TY, after appending a single column of one is partially redundant.

Finally, the random feature sets exhibit overlapping features. The number of fine-grained redundancies can grow significantly higher than this in real-world complex ML pipelines.

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



To address this redundancy, we introduce LIMA framework. Our work is based on the observation that lineage tracking enables the reuse of intermediates, model versioning, and debugging. LIMA is a framework for fine-grained lineage tracing and reuse inside ML systems. We employ low-overhead lineage tracing of individual operations, and full and partial reuse across the program hierarchy. We integrated LIMA into Apache SystemDS. However, the underlying techniques are fully or partially applicable for all types of ML Systems.

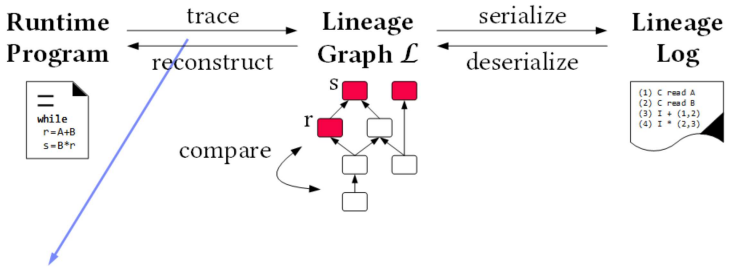
Lineage Tracing

(Key Operations and Lineage Deduplicaton)

We first describe efficient means to lineage tracing and the key operations.

7

Basic Lineage Tracing



Lineage Tracing Lifecycle

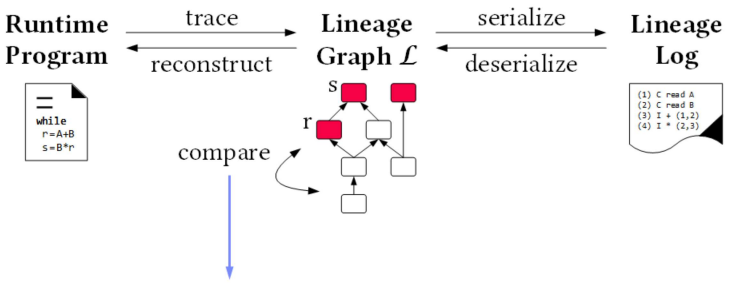
- **a) Efficient Lineage Tracing**
  - Trace lineage of logical operations for **all live variables**
  - Tracing of inputs, literals, and **non-determinism**
  - **Immutable** lineage DAG
  - Execution context maintains `LineageMap` that **maps live variable names to lineage items**

In this figure, we see the lifecycle of lineage tracing with the key operations. During runtime of a linear algebra program, we maintain lineage DAGs for all live

Variables. The immutable lineage DAG is then incrementally built as we execute runtime instructions. We maintain a LineageMap that maps live variable names to lineage items. Before executing an instruction, we obtain the lineage items for the instruction outputs, and update the lineage map. For non-deterministic operations like rand and sample, we include a system-generated seed in the corresponding lineage items.

8

Basic Lineage Tracing



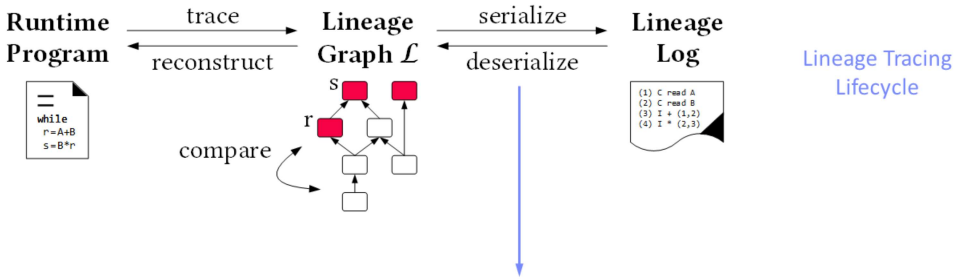
Lineage Tracing Lifecycle

- **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

A key operation is the comparison of lineage DAGs for equivalence. For this purpose, lineage items implement `hashCode()` and `equals()`. The hash code is computed as a hash over the hashes of the opcode, data item, and all inputs. The hash code is materialized in the immutable lineage items. Second, the equal check returns true if the opcode, data item, and all inputs are equivalent. In order to handle large DAGs, we use non-recursive, queue-based function implementations.

9

Basic Lineage Tracing

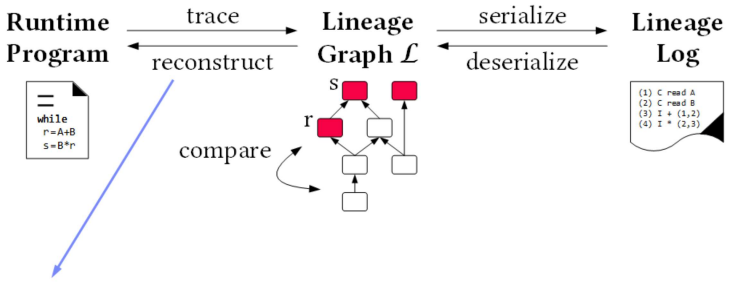


Lineage Tracing Lifecycle

- **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

Users may obtain the lineage of a variable in one of two ways, either by calling the built-in function lineage, or for every write, we also write the corresponding lineage. Serialization unrolls the DAG and write a single text line per lineage item. The lineage log can be deserialized back into a lineage DAG.

Basic Lineage Tracing



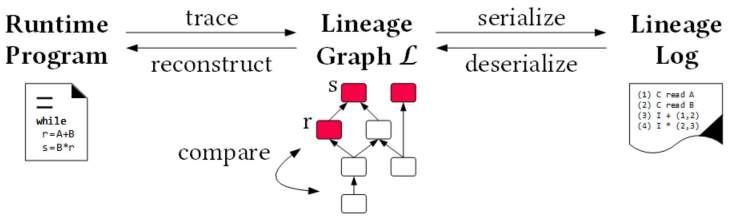
Lineage Tracing Lifecycle

- **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))))`

A very important utility for debugging is the ability to reconstruct a runtime program from a lineage DAG. The reconstructed program computes the exact same intermediates as the original program.

11

Basic Lineage Tracing



Lineage Tracing Lifecycle

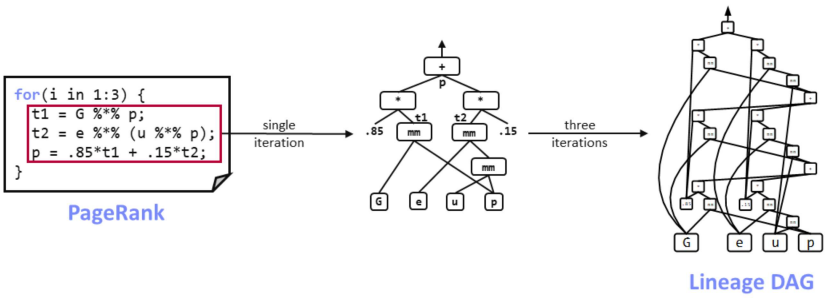
The entire lifecycle of lineage tracing with the key operations is very valuable as it **simplifies testing, debugging and reproducibility**.

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

12

Lineage Deduplication

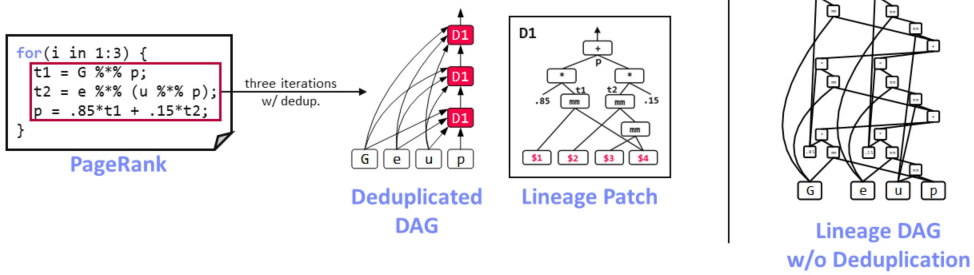
- **Problem**
  - Very **large lineage DAGs** for mini-batch training (repeated execution of loop bodies)
  - NN training w/ 200 epochs, batch-size 32, 10M rows, 1K instructions → 4TB → **4GB w/ deduplication**



A challenge of fine-grained lineage tracing is very large lineage DAGs, especially in mini-batch training scenarios. Large lineage DAGs originate from the repeated execution of code paths in loops and functions. In the figure, we see a script of pagerank graph algorithm. G is a large graph representing the linked websites, and p is the iteratively updated pagerank of individual sites. A single iteration of the loop body produces a DAG like the one on the right. Then this DAG repeats for every iteration and form a large lineage graph after completion of just 3 iterations. In another example of DNN training with 200 epochs, batch-size 32, 10 million rows and 1000 instructions per iteration , lineage DAG can grow up to 4 terabytes. We address this issue with a new concept of lineage deduplication, which can reduce the size to 4 gigabytes.

13

Lineage Deduplication



PageRank

Deduplicated DAG

Lineage Patch

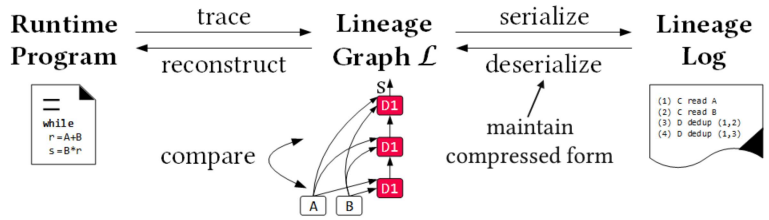
Lineage DAG w/o Deduplication

- **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

The basic idea of lineage deduplication is to eliminate these repeated patterns. Conceptually, we trace each independent control flow path once, store the sub-DAGs as patches, and refer them via a single lineage item. In the figure, we see the full lineage DAG after deduplication. A single lineage item, D1 is added per iteration, where D1 points to a lineage patch.

On entering a last-level loop, we count the distinct control paths. During runtime, we follow basic lineage tracing, but additionally we track the taken path using a bitvector. After each iteration, we save the patch for each live variable and add a single lineage item to the global DAG. Once lineage patches are created for all distinct paths, we stop tracing.

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**

We extended deduplication logic to integrate the key features of basic lineage tracing. To address non-determinism, such as dropout layers in DNN architecture, we add the seeds as literal inputs to the single dedup lineage items. Similarly, to avoid expansion of a deduplicated DAG, we extended equal to compare regular and deduplicated DAGs, and program reconstruction logic to compile lineage patches as functions.

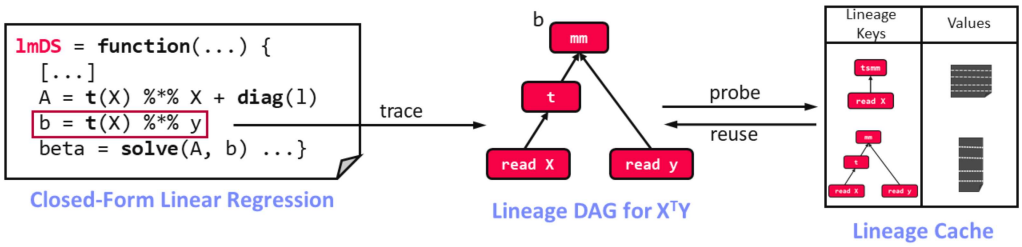
Lineage-based Reuse

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

As mentioned before, the lineage of an intermediate can uniquely identify the intermediate. We leverage this characteristic to eliminate fine-grained redundancy.

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()`



Closed-Form Linear Regression

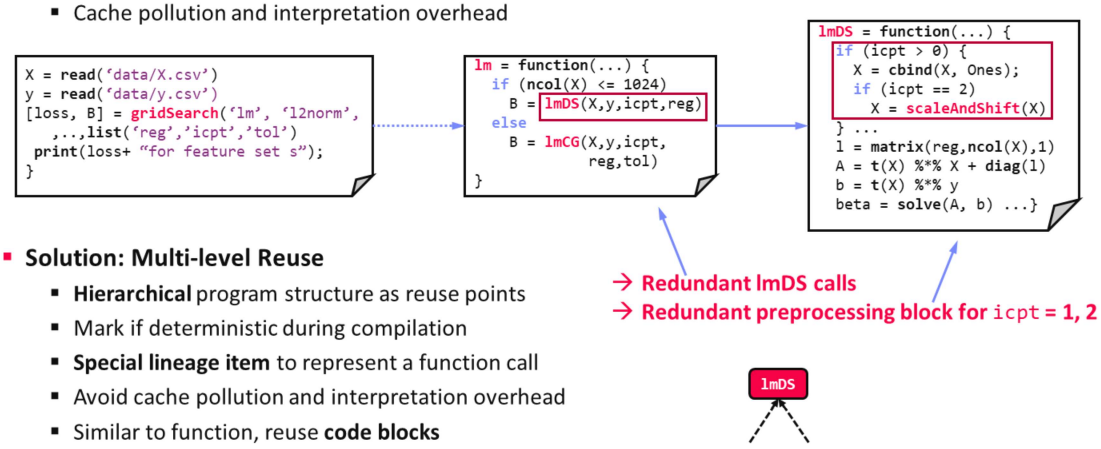
Lineage DAG for X^TY

Lineage Cache

As a foundation of lineage-based reuse, we establish a cache of intermediates. In the core of the cache, there is a hashmap that maps lineage traces to in-memory data objects. Returning to our running example. We trace lineage before executing an instruction, and use the obtained lineage trace to probe the cache. If available in the cache, we skip the instruction, otherwise we execute and store the result in the cache for future reuse.

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**

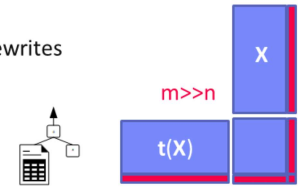
→ **Redundant lmDS calls**
→ **Redundant preprocessing block for** `icpt` **= 1, 2**

Basic instruction-level reuse eliminates fine-grained redundancy. However, this approach fails to remove coarse-grained redundancy, such as redundant function calls, redundant block of code. In our running example, we see 2 such redundancies. First, lmDS is called repetitively, sometimes with the same arguments. Second, the preprocessing block inside lmDS stays the same for intercept 1 and 2.

As a solution, we augment the basic reuse with multi-level reuse. Our basic idea is to use hierarchical program structures of functions and control flow blocks as probing and reuse points. During compilation, we declare a function a reuse candidate if the function doesn't contain any non-deterministic calls. In runtime, we construct a special lineage item for a function and use it as a key in the lineage cache. This way we are able to reuse function calls with the same inputs. Moreover, multi-level reuse over instruction-level reuse avoids cache pollution and interpretation overhead. We apply a similar logic to reuse control flow blocks.

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
- **Example Rewrites**
  - #1 `rbind(X, ΔX)Y → rbind(XY, ΔXY);`
  - #2 `Xcbind(Y, ΔY) → cbind(XY, XΔY)`
  - #3 `dsyrk(cbind(X, ΔX)) → rbind(cbind(dsyrk(X), XᵀΔX), cbind(ΔXᵀX, dsyrk(ΔX)))`
  - ...




```
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) ...}
```

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

where, dsyrk(X) = X^TX

We have come across use cases in ML pipelines, where operations have partially overlapping computations. Full reuse fails to eliminate these redundancies. Here is an example. Steplm is a stepwise forward feature selection algorithm. Steplm repetitively appends an additional feature to a global set of features and call linear regression to find the next best column. Repetitive execution of X^TX can be avoided by forming a compensation plan over the previously computed and cached outputs. Partial rewrites are pattern-based and written as an ordered list of source-target patterns. If full reuse is not possible, we probe for the partial reuse. If the operation matches with a source pattern, and the components of the target pattern are available in the lineage cache, then we construct an operation DAG over the cached items. We then compile the DAG and execute real instructions to obtain the result. Our existing rewrites are based on real use cases, and patterns with append, indexing, with expensive matrix multiplications, where the input matrices are larger than a certain heuristic-defined size.

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 & 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**]

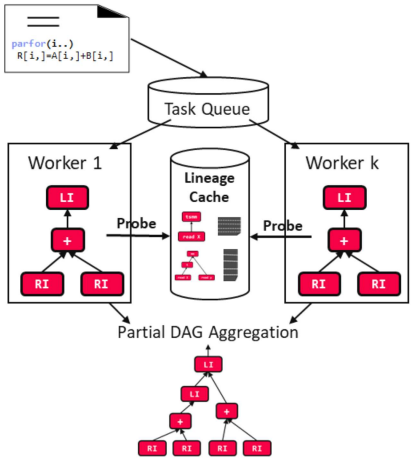
- **Eviction Policies**
  - Determine **order of eviction**
  - **LRU**: orders by normalized last access time
    → Pipelines with temporal reuse locality
  - **DAG-Height**: orders by depth of DAG (deep lineage traces have less reuse potential)
    → Mini-batch scenario. Reuse across epochs
  - **Cost&Size**: 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 **Default Policy**

Cache eviction ensures that the size of the cached objects does not exceed the configured budget. Our eviction logic spills cached objects to disk if the re-computation time of the exceeds the estimated I/O time. As a basis of our eviction policies, we collect various statistics per cache entry. On entering the cache, we obtain the execution time, the height of the operation DAG and the size of the intermediate. During reuse, we record the last access timestamp and the number of cache hits and misses. We estimate the disk I/O as a moving average of measured read, write time.

Similar to previous work, we use runtime eviction strategies. We have 3 eviction policies. These policies determine the order of eviction. Policy LRU orders the objects by the last access time. LRU performs well in pipelines with temporal reuse locality. Then DAG-Height assumes that deep lineage traces have less reuse potential, and orders accordingly. This works better in mini-batch scenarios with reuse potential across epochs. Finally, cost&size preserves objects with high cost to size ratio. It orders the objects by cost, size ratio, scaled by the number of hits and misses. Cost&size performs well in a wide variety of scenarios, and hence cost&size is our default choice.

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

We extended LIMA to integrate with modern ML systems, and take advantage of the widely used features and components. Modern ML systems provide means of task-parallel loops, for tasks like hyperparameter tuning. In a multi-threading scenario we trace lineage in a worker-local manner, and merge the results after completion. To ensure reuse in task-parallel scenarios, we enforce thread-safety on lineage cache. For redundant operations across tasks, while one task executes, others wait on an empty lineage item. That way we ensure reuse across tasks.

Operator fusion via code generation is crucial for performance and is commonly offered by ML systems. Fusion loses operator semantics, thus does not allow lineage tracing. To overcome this, we construct lineage patches during compilation, and store them in a dictionary. Later during runtime, we expand the lineage DAG by these patches. Similar to regular instructions, we also reuse the intermediates produced by the fused operators.

Lineage-based reuse in runtime is valuable, however, a ML system compiler can improve aspects of lineage cache and create additional reuse opportunities. We extended our compiler to unmarks intermediates for caching with no potential for reuse. This optimization helps to avoid cache pollution and unnecessary probing.

We further added several reuse-aware rewrites which prefer patterns that create additional reuse opportunities. These rewrites are placed in both compiler and runtime re-compiler.

Experiments

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

In this section, we will discuss a few snapshots of the end-to-end experiments. More experiments and micro bencmarks can be found in the paper.

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.

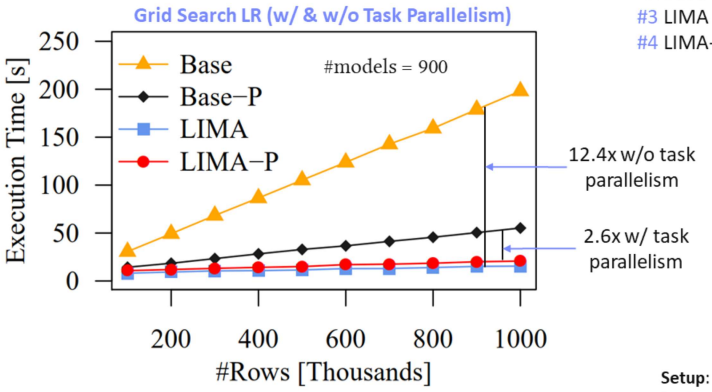
- **Workloads**

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

We compare LIMA against state-of-the-art ML systems, such as Tensorflow (2.3), Apache SystemDS, Scikit-learn, and coarse-grained reuse-based systems, such as Helix. We use real datasets, however, lineage-based reuse is largely independent of data skew, and more workload dependent. We evaluated a variety of ML pipelines with different characteristics, covering both traditional ML and Neural Network workloads.

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

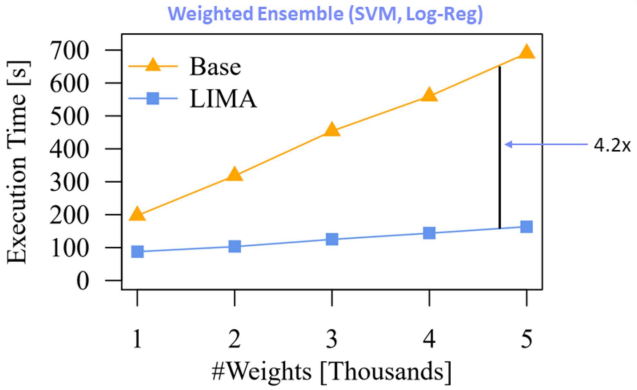
Setup: Hadoop cluster with each node having a single AMD EPYC 7302 CPUs (16/32 cores) and 128 GB DDR4 RAM

This experiment shows the result of our running example of gridsearch hyperparameter optimization over 900 models. Here we compare LIMA against SystemDS without lineage tracing and reuse. We also use task-parallel loop for hyperparameter tuning to better understand the impact of reuse on task-parallelism. We see improvements of 12.4x without task-parallelism, and 2.6x with task-parallel loop. This huge improvement comes from various multi-level and partial reuse of linear regression operations as discussed before.

Experiments

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

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



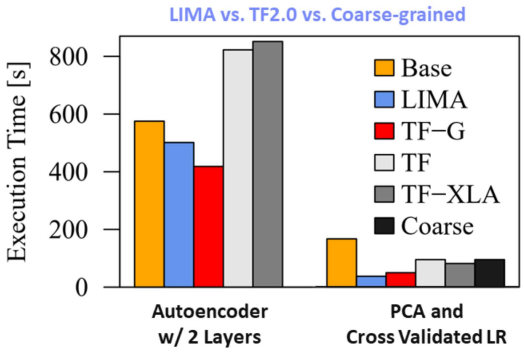
Large improvements due to **fine-grained redundancy elimination**

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

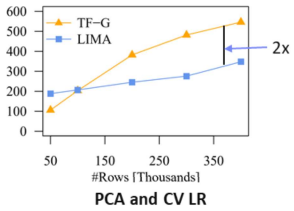
In this experiment we use a weighted ensemble of models. We first train 3 SVM and 3 logistic regression models, and then optimize the ensemble weights via random search. The scope for reuse is limited for iterative SVM and logistic regression training. However, we see a 4.2x improvement due to reused matrix multiplication in the computation of weighted class probabilities.

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

Additionally, we also compare LIMA against TensorFlow, Scikit-learn and Helix. In the first scenario, we use an Autoencoder with 2 hidden layers. We build a feature wise preprocessing map including normalization, binning, recoding and one-hot encoding, and apply this map batch-wise in each iteration. Even though Autoencoder has no reuse opportunities, LIMA performs 15% better over default SystemDS by reusing batch-wise preprocessing. TensorFlow, being a specialized system for mini-batch training, performs slightly better than LIMA in graph mode, whereas eager mode is substantially slower. Our implementation allows Tensorflow to build a single graph, which also allows a form of reuse. In the 2nd scenario, we have PCA and linear regression with cross validation. First we vary number of features for PCA, and then we vary regularization hyperparameter for regression. Coarse-grained systems shows improvement by using top-level PCA results. However, LIMA performs 3.7x better than coarse-grained due to additional fine-grained reuse. LIMA is 25% faster than Tensorflow graph due to partial reuse across folds. For larger datasets, Tensorflow runs out of memory, likely because the global graph keeps all the intermediates in memory, and misses eviction mechanism.

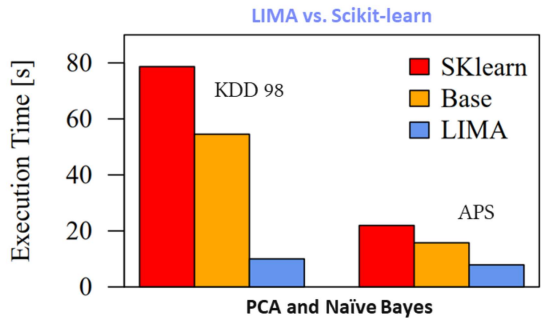
Experimental Evaluation

26

Experiments, cont.

- **ML Systems Comparison**

Baselines:
#1 SKlearn = Scikit-learn
#2 Base = **SystemDS** default config.
#3 LIMA = Lineage tracing and reuse



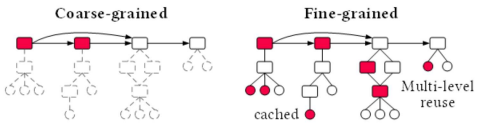
Competitive baseline performance against state-of-the-art ML Systems

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

In another experiment with hyperparameter tuning for PCA and Naïve Bayes, we see LIMA performs up to 8x better than Scikit-learn by reusing PCA intermediates and aggregate operations in Naïve Bayes.

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

To summarize, we introduce LIMA, a framework for fine-grained lineage tracing and reuse. LIMA presents multi-level lineage tracing with deduplication for loops, and seamlessly supports advanced features of modern ML systems. Compiler-assisted full and partial reuse allow removing redundancy at different levels of pipeline hierarchy. In conclusion, as the complexity of ML pipelines increases, increasing redundancy is inevitable and difficult to address by library developers. Compile time Common Subexpression Elimination can only be partially useful in removing fine-grained reduncancy due conditional control flow. A runtime-based lineage cache is proved effective to overcome these challenges. Interesting future work includes combine lineage cache with persistent materialization with multi-tenant and collaborative environment, support for distributed and multi-device caching, such as GPU intermediates, and extend lineage for model debugging and fairness constraints.