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

Arnab Phani
Graz University of Technology

Benjamin Rath
Graz University of Technology

Matthias Boehm
Graz University of Technology

ABSTRACT

Machine learning (ML) and data science workflows are 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. The repetitive nature of these workflows, and their hierarchical composition from building blocks exhibits high computational redundancy. Existing work addresses this redundancy with coarse-grained lineage tracing and reuse for ML pipelines. This approach allows using existing ML systems, but views entire algorithms as black boxes, and thus, fails to eliminate fine-grained redundancy and to handle internal non-determinism. In this paper, we introduce LIMA, a practical framework for efficient, fine-grained lineage tracing and reuse inside ML systems. Lineage tracing of individual operations creates new challenges and opportunities. We address the large size of lineage traces with multi-level lineage tracing and reuse, as well as lineage deduplication for loops and functions; exploit full and partial reuse opportunities across the program hierarchy; and integrate this framework with task parallelism and operator fusion. The resulting framework performs fine-grained lineage tracing with low overhead, provides versioning and reproducibility, and is able to eliminate fine-grained redundancy. Our experiments on a variety of ML pipelines show performance improvements up to 12.4x.

ACM Reference Format:
Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452788

1 INTRODUCTION

Machine Learning (ML) and data science have profound impact on many applications in practice. In the past, ML systems primarily focused on efficient model training and prediction. However, there is a trend toward systems support for ML pipelines and the entire data science lifecycle. Systems like KeystoneML [89], Amazon SageMaker [62], Scikit-learn [79], SystemDS [15], and TensorFlow TFX [11] provide abstractions for data integration, validation, and augmentation, feature extraction and engineering, model selection and hyper-parameter tuning, model training and prediction, as well as model debugging. Integrating these abstractions into ML systems is compelling because state-of-the-art data integration and cleaning largely rely on ML models [37]. However, complex ML pipelines create challenges regarding reproducibility and computational redundancy, which can be addressed with data provenance.

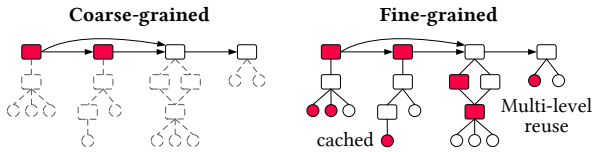


Figure 1: Coarse- and Fine-grained Lineage-based Reuse (for ML pipelines, utilizing hierarchically composed building blocks).

Data Provenance in ML Systems: Data provenance captures the origin and creation of data for understanding why and how query results were created [27, 42, 92]. Similarly, lineage—in terms of logical data transformations—has also been used for low-overhead fault tolerance in data-parallel frameworks like Apache Spark [105]. Recently, these concepts were adopted in ML system prototypes such as MISTIQUE [97], HELIX [103], Alpine Meadow [86], and the Collaborative Optimizer (CO) [34] for debugging and reusing intermediates. Existing approaches rely on coarse-grained lineage tracing at the level of ML pipelines and their top-level steps as shown in Figure 1-left. This black-box view of individual preprocessing steps, feature engineering, hyper-parameter tuning, and ML algorithms allows using existing—and rapidly evolving—ML systems. Unfortunately though, this approach fails to detect internal non-determinism and fine-grained redundancy. Exposing the internals of composite primitives at pipeline level is possible but requires a reimplementation of such primitives.

Problem of Non-Determinism: Many ML primitives are non-deterministic, so multiple runs with the same inputs do not yield the same results. Examples are (1) ML algorithms with randomly initialized models, (2) random reshuffling of matrices for mini-batch algorithms, cross validation, data partitioning, and splitting, (3) drop-out layers for regularization in deep neural networks (DNNs), and (4) basic randomized operations like `rand` or `sample`. Interestingly, recent work has found significant impact of random seeds on the model accuracy of cutting-edge DNNs [36]. When computing lineage for high-level primitives, this internal non-determinism—unless exposed via seed parameters—quickly becomes invisible. Externally encoding this metadata is possible but again defeats the purpose of being independent of underlying ML systems and their implementation. Therefore, non-determinism limits the use of coarse-grained lineage for versioning, reproducibility, and reuse.

Problem of Unnecessary Redundancy: The repetitive nature of exploratory data science processes, and increasingly complex, hierarchically composed ML pipelines and workflows, create high

SIGMOD '21, June 18–27, 2021, Virtual Event , China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452788

computational redundancy. While coarse-grained reuse can eliminate redundancy of identical steps, it fails to eliminate fine-grained redundancy. First, existing ML libraries and systems often compose high-level primitives from smaller building blocks to ensure reuse and consistency. For example, Scikit-learn [79] allows creating custom pipelines (via `make_pipeline()`) from sequences of preprocessing steps and ML algorithms, and then feed these pipelines into high-level primitives like `GridSearchCV`. Since this composition is unaware of independent operations and substeps, (and hand-optimized primitives for all combinations are infeasible), we end up with unnecessary redundancy. Unfortunately, in general-purpose programming languages, this redundancy is very difficult to correctly detect and eliminate. Second, entire ML algorithms might be repeatedly used inside feature selection and cross validation primitives, which show an orthogonal type of partial redundancy due to incrementally added features (feature selection), removed features (debugging), or overlapping fold compositions.

LIMA Framework Overview: Our LIMA framework overcomes these problems by a novel concept of *fine-grained* lineage tracing and reuse *inside* ML systems, as shown in Figure 1-right. We maintain a lineage DAG (directed acyclic graph) for all life variables during runtime of an ML program. Nodes represent executed operations—including parameters that make them deterministic (e.g., system-generated random seeds)—and edges are data dependencies. This DAG is recursively built, while executing conditional control flow and operations. The lineage of a variable exactly identifies an intermediate result, and can then be accessed, stored, and used to reproduce this intermediate. In order to eliminate unnecessary redundancy, we further leverage the lineage as keys in a reuse cache for full and partial reuse, with compensations for partial reuse. Figure 1 shows the resulting reuse opportunities at the granularity of individual operations, control-flow blocks, and functions.

Contributions: Our main contribution is the LIMA framework for efficient, fine-grained lineage tracing and reuse, implemented in Apache SystemDS[1] [15] as a representative ML system. Key ideas of this approach, beyond state-of-the-art, are (1) multi-level lineage tracing and reuse, (2) full and partial reuse of intermediates, and (3) a robust integration with ML system internals such as task parallelism and operator fusion. Our technical contributions are:

- *ML Systems Background:* To aid understanding, we provide the necessary background of ML system internals, and discuss different sources of redundancy in Section 2.
- *Lineage Tracing:* We then define the concept of Lineage DAGs, and discuss multi-level lineage tracing in Section 3. This discussion also includes the deduplication of lineage traces for functions, loops, blocks, and fused operators.
- *Lineage-based Reuse:* Leveraging these lineage traces, we introduce techniques for full and partial reuse of intermediates in Section 4. This reuse infrastructure relies on compiler-assisted, cost-based runtime caching, tailor-made eviction policies, and thread-safe access in task-parallel programs.
- *Experiments:* Finally, we report on extensive experiments in Section 5 that show low overhead lineage tracing, and significant performance improvements, compared to baselines like TensorFlow [1] and HELIX [103], on various ML workloads.

[1]The source code is available at https://github.com/apache/systemds.

2 BACKGROUND AND PRELIMINARIES

This section introduces our running example, necessary background of ML system internals, as well as common types of redundancy.

2.1 Running Example

Example 1 shows a user-level example ML pipeline—written in SystemDS' DML scripting language with R-like syntax [15]—which we use as a running example throughout this paper.

Example 1 (GridSearch LM). *We read a feature matrix* **X** *and labels* **y**, *and extract 10 random subsets of 15 features. For each feature set, we tune the linear regression (lm) hyper-parameters regularization, intercept, and tolerance via grid search and print the loss.*

```
1: X = read('data/X.csv'); # 1M x 100
2: y = read('data/y.csv'); # 1M x 1
3: for( i in 1:10) {
4:   s = sample(15, ncol(X));
5:   [loss, B] = gridSearch('lm', 'l2norm',
       list(X[,s],y), list('reg','icpt','tol'),...);
6:   print("Feature set ["+toString(s)+"]: "+loss);
7: }
```

High-level primitives like `gridSearch` *and* `lm` *are themselves script-based built-in functions and imported accordingly. Below functions show their key characteristics in simplified form:*

```
01: gridSearch = function(...) return(...) {
02:   HP = ... # materialize hyper-parameter tuples
03:   parfor( i in 1:nrow(HP) ) { # parallel for
04:     largs = ... # setup list hyper-parameters
05:     rB[i,] = t(eval(train, largs));
06:     rL[i,] = eval(score, list(X,y,t(rB[i,])));
07: } }
08: lm = function(...) return(...) {
09:   if (ncol(X) <= 1024)  # select closed-form
10:     B = lmDS(X, y, icpt, reg, verbose);
11:   else                  # select iterative
12:     B = lmCG(X, y, icpt, reg, tol, maxi, verbose);
13: }
14: lmDS = function(...) return(...) {
15:   if (icpt > 0) {
16:     X = cbind(X, matrix(1,nrow(X),1));
17:     if (icpt == 2)
18:       X = scaleAndShift(X); # mu=0,sd=1
19:   } ...
20:   A = t(X) %*% X + diag(matrix(reg,ncol(X),1);
21:   b = t(X) %*% y;
22:   beta = solve(A, b);
23: }
24: lmCG = function(...) return(...) {
25:   if (icpt > 0) {
26:     X = cbind(X, matrix(1,nrow(X),1));
27:     if (icpt == 2)
28:       X = scaleAndShift(X); # mu=0,sd=1
29:   } ...
30:   while (i<maxi & norm_r2>norm_r2_tgt) {
31:     q = t(X) %*% (X %*% ssX_p); ...
32:     p = -r + (norm_r2 / old_norm_r2) * p;
33: } }
```

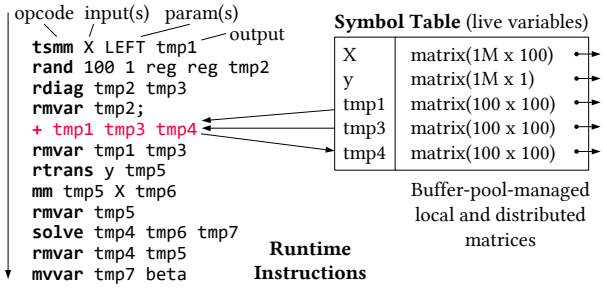


Figure 2: Operator Scheduling and Runtime Plans.

The `gridSearch` *function enumerates and materializes all hyper-parameter combinations* **HP** *of the passed parameters and value ranges, and invokes training (*`lm`*) and scoring (*`l2norm`*) functions to find the best model and loss. The* `lm` *function in turn dispatches—based on the number of features—either to a closed-form method with $O(m \cdot n^2 + n^3)$ complexity (*`lmDS`*); or an iterative conjugate-gradient method with $O(m \cdot n)$ per iteration (*`lmCG`*), which performs better for many features as it requires $\leq n$ iterations until convergence.*

2.2 ML Systems Background

There is a variety of existing ML systems. Relevant for understanding this paper, are especially the underlying techniques for program and DAG compilation, and operator scheduling [17]. Here, we focus primarily on lazy evaluation and program compilation.

Program/DAG Compilation: We distinguish three types of compilation in contemporary ML systems: (1) interpretation or eager execution, (2) lazy expression or DAG compilation, and (3) program compilation. First, interpretation as used in R, PyTorch [78], or Python libraries like NumPy [95] or Scikit-learn [79] execute operations as-is and the host language (e.g., Python) handles the scoping of variables. Second, systems like TensorFlow [1], OptiML [90], and Mahout Samsara [85] performing lazy expression evaluation that lazily collects a DAG of operations, which is optimized and executed on demand. Some of these systems—like TensorFlow or OptiML—additionally provide control flow primitives, integrated in the data flow graph. Here, the host language still interprets the control flow, and thus, unrolls operations into a larger DAG. However, recent work like AutoGraph [68] automatically compiles TensorFlow control flow primitives. Only bound output variables leave the scope of expression evaluation. Third, program compilation in systems like Julia [12], SystemML [14], SystemDS [15], and Cumulon [48] compiles a script into a hierarchy of program blocks, where every last-level block contains DAGs of operations. Accordingly, control flow and variable scoping is handled by the ML system itself. Despite the large optimization scope of lazy expression evaluation and program compilation, unnecessary redundancy cannot be fully eliminated via code motion and common subexpression elimination (CSE) because the conditional control flow is often unknown.

Operator Scheduling: Given a DAG of operations of an expression or program block, operator scheduling then determines an execution order of the individual operations, subject to the explicit data dependencies (i.e., edges) of the data flow graph. The two predominant approaches are sequential and parallel instruction streams. First, a sequential instruction stream linearizes the DAG—in depth- or breadth-first order—into a sequence of instructions that is executed one-at-a-time. For example, Figure 2 shows a plan of runtime instructions in SystemDS for lines 21-23 of Example 1. A symbol table holds references to live variables and their metadata. Instructions are executed sequentially, read their inputs from a variable map (a.k.a. symbol table), and put their outputs back. Such a serial execution model—as used in PyTorch [78] and SystemML [14, 16]—is simple and allows bounding the memory requirements. Second, parallel instruction streams—as used in TensorFlow [1]—leverage inter-operator parallelism: when all inputs of an operation are available, this operation is enqueued for parallel execution. This execution model offers a high degree of parallelism (for many small operations) but makes memory requirements less predictable.

2.3 Sources of Redundancy

We can now return to our running example and discuss common sources of fine-grained redundancy.

Example 2 (GridSearch LM Redundancy). *The user script from Example 1 with a $1M \times 100$ feature matrix* $\mathbf{X}$ *and three hyper-parameters (*`reg`, `icpt`, `tol` *with 6, 3, and 5 values) exhibits multiple sources of redundancy. First, since* $\mathbf{X}$ *has 100 features, all calls to* `lm` *are dispatched to* `lmDS` *and thus, one of the hyper-parameters (*`tol`*) is irrelevant and we train five times more models than necessary. Second, evaluating different λ parameters (*`reg`*) for* `lmDS` *exhibits fine-grained operational redundancy. The core operations $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are independent of* `reg` *and thus, should be executed only once for different λ. Third, both* `lmDS` *and* `lmCG` *have the same pre-processing block, and for 2/3 of* `icpt` *values, we perform the same* `cbind` *operation, which is expensive because it creates an intermediate larger than* $\mathbf{X}$*. Fourth, appending a column of ones does not require re-executing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Instead we can reuse these intermediates and augment them with colSums($\mathbf{X}$), sum($\mathbf{y}$) and nrow($\mathbf{X}$). Similarly, the random feature sets will exhibit overlapping features whose results can be reused.*

Types of Redundancy: Existing work performs reuse for coarse-grained sub-tasks in ML pipelines [34, 86, 89, 97, 103, 107]. Generalizing upon the previous example, we further extend this to common types of *fine-grained* redundancy:

- *Full Function or Block Redundancy:* At all levels of the program hierarchy, there is potential for full reuse of the outputs of program blocks. This reuse is a form of function memoization [29], which requires deterministic operations.
- *Full Operation Redundancy:* Last-level operations can be reused for equivalent inputs, given that all non-determinism (e.g., a system-generated seed) is exposed from these operations and cast to a basic input as well.
- *Partial Operation Redundancy:* Operation inputs with overlapping rows or columns further allow reuse by extraction from—or augmentation of—previously computed results.

Together, these different types of redundancy motivate a design with (1) fine-grained lineage tracing, (2) multi-level, lineage-based reuse, and (3) exploitation of both full and partial reuse.

Applicability in ML Systems: Fine-grained lineage tracing and reuse is applicable in ML systems with eager execution, lazy evaluation, and program compilation. In contrast, multi-level tracing, deduplication, and reuse require access to control structures and thus, are limited to systems with program compilation scope.

3 LINEAGE TRACING

As a foundation of lineage-based reuse, we first describe efficient means to lineage tracing and key operations. Lineage graphs may get very large though, especially for mini-batch training. For this reason, we introduce the idea of lineage deduplication for loops and functions. Finally, we discuss design decisions and limitations.

3.1 Basic Lineage Tracing

During runtime of a linear algebra program, LIMA maintains—in a thread- and function-local manner—lineage DAGs for all live variables of this execution context. Figure 3 shows the lifecycle of such lineage information and key operations.

Definition 1 (Lineage DAGs). *A lineage DAG $\mathcal{L}$ is a directed, acyclic graph, whose nodes (or lineage items) represent operations and their outputs, and whose edges represent data dependencies. Lineage items consist of an ID, an opcode, an ordered list of input lineage items, an optional data string and hash, and a visited flag for memoization of processed sub graphs. Leaf nodes are literals or matrix creation operations (e.g.,* `read` *or* `rand`*), and multiple inner nodes might refer to the same inputs. Thus, the lineage DAG is a data flow graph, that encodes the exact creation process of intermediate results, without the computation that determined the control flow path.*

Lineage Tracing: The immutable lineage DAG for live variables is then incrementally built by lineage tracing as we execute runtime instructions (Figure 3, *trace*). Every execution context maintains a `LineageMap` that maps live variable names to lineage items (Figure 3, red root nodes), and caches literal lineage items. As a lean runtime integration, individual instructions—in a class hierarchy of instructions for local and distributed operations—implement a dedicated interface `LineageTraceable` for obtaining lineage items. Before[2] executing an instruction (integrated in `preprocessInstruction`), we obtain the lineage items for the instruction output(s) and update the lineage map. Special instructions like `mvvar` and `rmvar`—for renaming and removing variables—only modify the mapping of live variables to lineage items. For capturing non-determinism, we also modified selected runtime instructions, like `rand` or `sample`, to create system-generated seeds on `preprocessInstruction` for inclusion in the lineage items.

Comparisons: When working with multiple, potentially overlapping lineage DAGs, a key operation is the comparison of two lineage DAGs for equivalence or containment (Figure 3, *compare*). For this purpose, lineage items implement `hashCode()` and `equals()`, whose semantics are recursively defined. First, the hash code of a lineage item is computed as a hash over the hashes of the opcode, data item, and all inputs. As lineage DAGs are immutable, we cache the computed hash for every lineage item. Second, the equals check returns true if the opcode, data item, and all inputs are equivalent. In order to handle large DAGs, we use memoization to avoid redundant processing of sub-DAGs reachable over multiple paths, and non-recursive, queue-based function implementations.

Serialization and Deserialization: Users may obtain the lineage in two forms. First, a new `lineage(X)` built-in function returns the lineage DAG of variable **X** as a string. Second, for every write to a file `write(X,'f.bin')`, we also write the lineage DAG to a text file `'f.bin.lineage'`. Both cases require the serialization of the lineage DAGs (Figure 3, *serialize*). This serialization unrolls the lineage DAG in a depth-first manner, creating a text line per lineage item. Inputs are represented via IDs and memoization ensures that every item is serialized once in the lineage log. To handle large DAGs, we again use a non-recursive implementation with stack-based data structures. The lineage log can be deserialized back into a lineage DAG (Figure 3, *deserialize*) by processing the lineage log line-by-line. For every line, we obtain input items from a lookup table, create the lineage item, and store it in the lookup table.

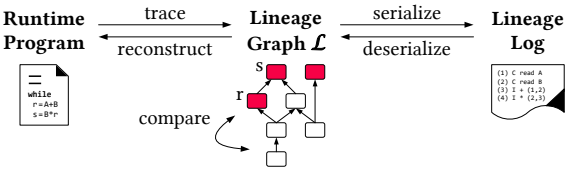


Figure 3: Lineage Tracing Lifecycle and Operations.

Re-computation from Lineage: Additionally, we provide a utility for generating a runtime program from a lineage DAG (Figure 3, *reconstruct*) that computes—given the same input—exactly the same intermediate. In contrast to the original program, the reconstructed program does not contain control flow but only the operations for computing the output. The entire lifecycle from tracing, over serialization and deserialization, to the re-computation by lineage is very valuable as it simplifies testing, debugging, and reproducibility as illustrated by the following example.

Example 3 (Debugging with Lineage). *Let us share a debugging story from practice, which motivated the lineage support in SystemDS. Users deployed a sentence classification pipeline in production, noticed differences in results compared to the development setup, and reported this as a blocking issue. We reproduced the setup, spent nights debugging it up to round-off errors of different degrees of parallelism, and yet, still could not reproduce it. Finally, we found that the modified deployment infrastructure passed arguments incorrectly, making the pipeline use default parameters. With lineage support, such multi-person debugging efforts become much simpler: lineage logs can be exchanged, compared, and used to reproduce results.*

3.2 Lineage Deduplication

A new challenge of fine-grained lineage tracing are potentially very large lineage DAGs in use cases like mini-batch training. Consider an average lineage item size of 64 B, and training 200 epochs on a dataset of 10M rows, batch-size 32, and 1,000 instructions per iteration. The resulting lineage DAG would grow to 4 TB. We address this issue inside ML systems with a new concept of *lineage deduplication*, reducing the size to 4 GB in this example. Additionally, deduplication can remove the overhead of repetitive tracing.

Basic Idea: Large lineage DAGs originate from the repeated execution of code paths in loops and functions, which create repeated patterns in the lineage graph. The basic idea of lineage deduplication is to eliminate these repeated patterns in the lineage DAG as a form of compression. Conceptually, we extract lineage sub-DAGs called patches, store them once, and refer to these patches via a single lineage item. Since reactive deduplication (after lineage tracing)—similar to function outlining—is a hard problem and

[2]Lineage tracing *before* instruction execution facilities reuse as described in Section 4.

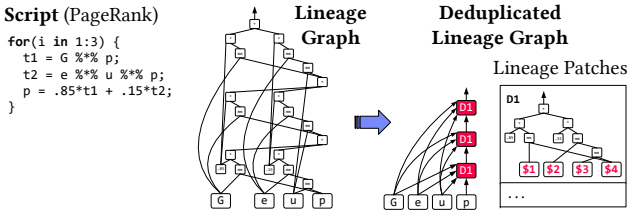


Figure 4: Example Lineage Deduplication for PageRank.

brittle, we perform proactive deduplication on entering last-level loops and functions. However, as the number of lineage patches is exponential in the number of branches, we use a *hybrid* design with proactive setup, and minimal runtime tracing.

Loop Deduplication Setup: On entering a last-level `for`, `parfor`, or `while` loop, we analyze the distinct control paths to aid deduplication. The distinct control paths are all possible execution paths (e.g., 2^3 paths for a sequence of three `if-else`-blocks), each with its own lineage patch. During setup, we count these paths in a single pass through the program, replicating the current set of traced paths at every branch. In this process, we also assign branch positions (IDs) and materialize these IDs in the `if-else` program blocks. For nested branches, the IDs are assigned in a depth-first order of the entire subprogram. Additionally, we obtain the inputs and outputs of the loop body from live variable analysis, and prepare an empty map of lineage patches but do not materialize these patches to avoid unnecessary setup for paths that are never taken.

Loop Deduplication Tracing: During iteration runtime, we trace temporary lineage DAGs. We first construct ordered placeholder items for the loop inputs and indexes. Additionally, we initialize a bitvector **b** for tracing the taken path, where bit $\mathbf{b}_i$ is set to the evaluated condition of branch i. We then execute the loop body, while performing basic lineage tracing and updating **b**. At the end of an iteration, we maintain the map of lineage patches and the global lineage DAG. The bitvector **b** represents the key of the lineage patch, and we keep the collected lineage DAG as a new patch if it does not yet exist. Finally, a single dedup lineage item—pointing to the lineage patch—is added onto the global lineage DAG. Once lineage patches are created for all distinct paths, we stop this on-demand lineage tracing, and only trace the taken control paths.

Example 4 (PageRank Loop Deduplication). *Figure 4 illustrates this concept of loop deduplication for a classical PageRank graph algorithm. On the left, we see the original script, where* **G** *is a sparse graph representing the linked websites, and* **p** *is the iteratively updated page rank of individual sites. When executing three iterations without deduplication, we get the lineage graph in the center with repeated substructures. In contrast, with loop deduplication, we have extracted one lineage patch with four inputs and one output, and add a single lineage item per iteration to the lineage graph.*

Function Deduplication: Similar to loop deduplication, we apply the same concept for functions that do not contain loops or other function calls. We again count the distinct control paths upfront, use the bitvector approach to trace the taken path, and add a single lineage item per function call to the lineage graph. Additional support for nested loops and function calls is interesting future work. We focused on last-level loops and functions, which offers a good tradeoff between simplicity and benefit of deduplication.

Handling of Non-Determinism: Coming back to our example of mini-batch training. Many DNN architectures contain `dropout` layers for regularization, which is a non-deterministic operation that generates new dropout masks in every iteration. Our approach to handling such non-determinism in the context of deduplication is to model the seeds as input placeholders of the lineage patch, trace these seeds like the control path bitvector, and add them as literal inputs to the single dedup item. Similarly, all functions are tagged as deterministic or non-deterministic during compilation.

Operations on Deduplicated Graphs: All basic lineage operations apply to deduplicated lineage graphs too. However, naïvely decompressing the lineage graph—by lookup of lineage patches and expansion—would defeat the purpose of deduplication. We alleviate this problem by two extensions. First, we *serialize* and *deserialize* the dictionary of lineage patches to preserve the deduplication for storage and transfer. We further extended the *compare* functionality to match normal and deduplicated sub-DAGs, by enforcing equal hashes for regular and dedup items, and resolving dedup items if needed. Second, program reconstruction would also cause expansion. Hence, on *reconstruct*, we compile the lineage patches into functions, and sequences of equivalent dedup items into loops.

3.3 Lineage Tracing for Advanced Features

Modern ML systems further provide advanced features such as (1) operator fusion, and (2) task-parallel for loops, which are both widely used and thus, important to integrate with lineage tracing.

Operator Fusion: Operator fusion via code generation is crucial for performance because it can avoid materialized intermediates [18, 30, 77], allow scan sharing and sparsity exploitation [18, 48], and kernel specialization for accelerators [8, 26, 83]. However, fusion loses the operator semantics and thus, does not allow lineage tracing. This limitation is problematic because it cuts the lineage trace into unusable pieces. Our approach is simple, yet effective. We construct the lineage patches of fused operators (with ordered placeholders) during compilation, and store them in a dictionary. During runtime, we expand the lineage graph by these lineage patches. Lineage now also enables new techniques such as de-optimizing fused operators and reuse-aware fusion.

Task-parallel Loops: Numerical computing frameworks like MATLAB [87], R [75], or Julia [12], and ML systems like TensorFlow [1] or SystemML [14, 19] provide means of task-parallel loops (e.g., for hyper-parameter tuning). Implementation details vary, but often multi-threaded and/or distributed workers are used. For ensuring isolation, we trace lineage in a worker-local manner, but individual lineage graphs share their common input lineage. Distributed operations leverage the *serialize* and *deserialize* operations to transfer lineage. The worker results are merged by taking their lineage roots and constructing a linearized lineage graph.

3.4 Limitations

The LIMA lineage tracing makes several tradeoffs. In the following, we discuss these design decisions and related limitations.

- *Immutable Files/RDDs:* We assume input files and RDDs are read-only (i.e., deterministic reads), which is a reasonable assumption and eliminates the need for data summarization.

- *No Capturing of Control Flow:* The lineage DAG represents the computation of an intermediate without the control path decisions. We made this choice because the original script is a more concise representation of the actual program.
- *Result Differences:* Despite handling non-determinism, reconstructed programs might produce slightly different results. Reasons include multi-threaded or distributed operations (aggregation orders), different environments (memory budgets, cluster size), and different artifacts (SW versions).

Our design focuses primarily on simplicity, efficiency, and robustness, which are key for leveraging lineage in many use cases like versioning, debugging, auto differentiation, and lineage-based reuse.

4 LINEAGE-BASED REUSE

The lineage of an intermediate carries all information to identify and recompute this intermediate. LIMA leverages this characteristic in a lineage-based reuse cache for eliminating fine-grained redundancy (see Section 2.3). Figure 5 gives an overview of our reuse approach. In this section, we describe (1) the lineage cache organization, and multi-level reuse of intermediates for functions, blocks, and operations, (2) partial reuse of operations with compensations, (3) cost-based eviction policies, (4) compiler-assisted reuse (e.g., rewrites), and (5) remaining limitations and future work.

4.1 Multi-Level Full Reuse

As the foundation of lineage-based reuse, we establish a cache that maps lineage traces to cached, in-memory data objects. Here, we describe a holistic design for a robust system integration.

Lineage Cache: The basic architecture of the lineage cache—as shown in Figure 5—comprises a hash map from lineage items (i.e., lineage traces of values) to cached values. These values can be matrices, frames, or scalars and are wrapped into lineage cache entries that hold additional metadata such as the data type, cache status, computation time, access timestamps, and eviction scores. The cache size is a configurable fraction of the maximum heap size (5% by default). Other configurations include the set of reusable instruction opcodes, the used eviction policy, and related parameters.

Full Reuse: During runtime, we then trace lineage *before* instruction execution, and use the obtained lineage item to probe the lineage cache for existing outputs. Full reuse refers to an operation-level reuse of a previously computed output in full, that is, without any compensations. If this probe succeeds, we obtain the cached value, put this value into the symbol table of live variables, and skip the instruction. If the value does not yet exist in the cache, we execute the instruction and additionally store its output(s) in the lineage cache. The probing leverages the *compare* functionality (via `hashCode()` and `equals()`) as described in Section 3.1. The time complexity of naïve `hashCode` and `equals` implementations are linear in the size of the lineage trace. However, due to materialized hash codes, we get constant complexity for constructing and hashing a new lineage item over existing lineage inputs, and using the hash codes for pruning[3]—in `equals` calls—works very well in practice. The reuse logic is integrated in the main instruction execution code path, which seamlessly applies to all instructions.

[3]Long lineage traces with repeated structure are prone to hash collisions due to integer overflows. We handle such overflows with separate hash functions.

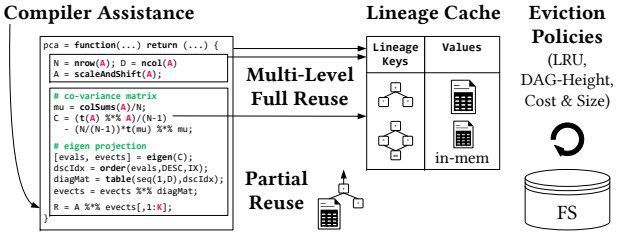


Figure 5: Overview Lineage-based Reuse.

In addition, making the set of cacheable instructions and data types configurable also avoids unnecessary cache pollution, and ensures correctness (e.g., for update-in-place indexing).

Multi-level Reuse: Our basic full reuse eliminates fine-grained redundancy at operation level in a robust manner. However, this approach is suboptimal for coarse-grained redundancy (e.g., reuse of entire functions) because it still requires a pass over all instructions and reuse of their cached outputs. Inspired by recent work on alternative probing strategies in CO [34] and pruning in HELIX [103], we augment the basic reuse by a multi-level reuse approach to avoid cache pollution and interpretation overhead. Our basic idea is to leverage the hierarchical program structure of functions and control flow blocks (see background in Section 2.2) as natural probing and reuse points. As pre-processing step, we determine and materialize if a given function or block is deterministic, i.e., does not include any non-deterministic operations or function calls. A block is similar to a function as it has specific inputs and outputs, which we obtain from live variable analysis. During runtime, we then construct a special lineage item that represents the function inputs, function call, and bundles all function outputs. If a deterministic function is called with the same inputs, we can thus, directly reuse its outputs; otherwise we execute the function and also bind cached outputs to the special lineage item. Multi-level reuse is orthogonal to lineage deduplication, but internally leverages shared infrastructure.

Example 5 (PCA Multi-level Reuse). *Figure 5 also illustrates this concept of multi-level reuse, where we first probe the entire* `pca` *(principal component analysis) function call. If we cannot reuse, we probe individual blocks (e.g., the block around* `scaleAndShift`*), and finally, individual operations (e.g.,* $\mathbf{A}^\top\mathbf{A}$ *as used in* `pca` *and* `lmDS`*).*

Task-parallel Loops: Similar to lineage tracing, supporting task-parallelism requires further lineage cache extensions. First, multi-threaded `parfor` workers concurrently probe and update the shared lineage cache. This concurrency requires a thread-safe lineage cache, which we ensure via latches and a careful implementation that keeps the critical sections small and prevents deadlocks. Second, we use lineage cache "placeholders" (empty lineage cache entries) to avoid redundant computation in parallel tasks. For example, consider the following hyper-parameter tuning loop:

```
1: parfor(i in 1:nrow(lambda))
2:   B[,i] = lmDS(X=X, y=Y, reg=lambda[i,1]);
```

For instance, k=32 iterations might run in parallel here, and during the first wave of iterations $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are not yet available for reuse. This redundancy matters because we should rather spend the parallelism in individual operations if needed. Hence, the first

thread puts a placeholder in the cache. Other threads then find this placeholder and block on obtaining its matrix, frame, or scalar until the first thread adds the computed value back to the placeholder.

4.2 Partial Operation Reuse

Beyond multi-level full reuse, LIMA also eliminates *partial operation redundancy* via partial reuse. Partial reuse refers to an operation-level reuse of a previously computed output, augmented by a compensation plan of reused or computed operations.

Partial Reuse: If full reuse is not possible, we quickly probe different partial reuse opportunities, and only if none applies, fallback to normal instruction execution. This probing for partial reuse evaluates an ordered list of rewrites of source-target patterns. If the current lineage item (before execution) matches a source pattern, and components of the target pattern are available in the lineage cache, we construct a *compensation plan* to compute the result. Similar to constant folding, we put reusable intermediates of the target pattern into a temporary symbol table, construct an operation DAG for the remaining operations, and then, compile and execute actual runtime instructions to obtain the results. The rewrites are hand-written and can include cost-based constraints. Our existing rewrites focus primarily on real use cases and patterns with `rbind`, `cbind`, and indexing in combination with matrix multiplications, column or row aggregates, and element-wise operations.

Example Rewrites: Our set of partial rewrites comprises 14 meta rewrites with internal variants. Below equation shows selected examples, where $\mathbf{X}\,\mathbf{Y}$ is a matrix multiply, $\odot$ and $+$ are element-wise multiply and addition, and $\mathbf{dsyrk}(\mathbf{X})$ is a shorthand for $\mathbf{X}^\top\mathbf{X}$.

$$\begin{aligned}
\mathbf{rbind}(\mathrm{X}, \Delta\mathrm{X})\,\mathrm{Y} &\rightarrow \mathbf{rbind}(\mathrm{X}\,\mathrm{Y}, \Delta\mathrm{X}\,\mathrm{Y})\\
\mathrm{X}\,\mathbf{cbind}(\mathrm{Y}, \Delta\mathrm{Y}) &\rightarrow \mathbf{cbind}(\mathrm{X}\,\mathrm{Y}, \mathrm{X}\,\Delta\mathrm{Y})\\
\mathrm{X}\,\mathbf{cbind}(\mathrm{Y}, \mathbf{1}) &\rightarrow \mathbf{cbind}(\mathrm{X}\,\mathrm{Y}, \mathbf{rowSums}(\mathrm{X}))\\
\mathrm{X}\,(\mathrm{Y}[, 1:k]) &\rightarrow (\mathrm{X}\,\mathrm{Y})[, 1:k]\\
\mathbf{dsyrk}(\mathbf{rbind}(\mathrm{X}, \Delta\mathrm{X})) &\rightarrow \mathbf{dsyrk}(\mathrm{X}) + \mathbf{dsyrk}(\Delta\mathrm{X})\\
\mathbf{dsyrk}(\mathbf{cbind}(\mathrm{X}, \Delta\mathrm{X})) &\rightarrow \mathbf{rbind}(\mathbf{cbind}(\mathbf{dsyrk}(\mathrm{X}), \mathrm{X}^\top\Delta\mathrm{X}),\\
&\qquad\qquad \mathbf{cbind}(\Delta\mathrm{X}^\top\,\mathrm{X}, \mathbf{dsyrk}(\Delta\mathrm{X})))\\
\mathbf{cbind}(\mathrm{X}, \Delta\mathrm{X}) \odot \mathbf{cbind}(\mathrm{Y}, \Delta\mathrm{Y}) &\rightarrow \mathbf{cbind}(\mathrm{X} \odot \mathrm{Y}, \Delta\mathrm{X} \odot \Delta\mathrm{Y})\\
\mathbf{colAgg}(\mathbf{cbind}(\mathrm{X}, \Delta\mathrm{X})) &\rightarrow \mathbf{cbind}(\mathbf{colAgg}(\mathrm{X}), \mathbf{colAgg}(\Delta\mathrm{X}))
\end{aligned}$$

Similar rewrites have been used for incremental linear algebra programs [72], linear algebra simplification rewrites [16, 39, 40, 54, 100], and hand-optimized cross validation primitives [58]. In contrast, we apply them during lineage-based reuse or recompilation. Application examples are `lm` and `scaleAndShift` with and without intercept, cross validation, as well as `stepLm` [15, 99], a feature selection algorithm that repeatedly appends additional features.

4.3 Cache Eviction

Lineage cache eviction ensures that the size of cached objects does not exceed the configured budget. Important aspects are the cache management, and dedicated cost-based eviction policies.

Cache Management: Given an absolute memory budget B (by default, 5% of heap size), intermediates of size smaller than B are subject to caching. On adding such an object o to the cache, we get its in-memory size $s(o)$ and trigger eviction if the cache size $S + s(o)$ exceeds B. Objects under eviction management (excluding empty placeholders), are then added to a priority queue $\mathbf{Q}$ that orders objects for eviction according to eviction-policy-specific scoring functions. An active eviction then pulls and evicts items in order until the budget constraint $S \leq B$ is met. The eviction of an object is either by deletion or by spilling to disk, where we only spill objects whose re-computation time exceed the estimated I/O time. We perform additional bookkeeping to allow identifying such spilled cache entries. With multi-level reuse, multiple cache entries might refer to the same object but exhibit different costs (e.g., function versus operation). In such scenarios, we defer the spilling of an entry group until all group entries are pulled for eviction.

Table 1: Eviction Policies and Scoring Functions.

Eviction Policy	Eviction Scoring Function
LRU	$\arg\min_{o\in\mathbf{Q}} \; T_a(o)/\theta$
DAG-Height	$\arg\min_{o\in\mathbf{Q}} \; 1/h(o)$
Cost & Size	$\arg\min_{o\in\mathbf{Q}} \; (r_h + r_m)\cdot c(o)/s(o)$

Statistics and Costs: As a basis for our eviction policies and user-facing statistics, we collect various statistics. On entering the cache, we obtain (1) the measured function or operation execution time $c(o)$ of cached objects, and (2) the height $h(o)$ of related lineage traces (distance from leaves). During cache usage, we further gather (3) the last access timestamp $T_a(o)$, and (4) the number of references (#hits r_h, #misses r_m). Additionally, we estimate (5) the spill (write) and restore (read) times of cached objects. We obtain the sizes in memory $s_m(o)$ and on disk $s_d(o)$ from the data characteristics, and scale $s_d(o)$ with expected read/write bandwidths for dense and sparse matrices, or frames. For adaptation to the underlying hardware, we adjust these expected bandwidths (starting heuristics) as an exponential moving average with the measured I/O times.

Eviction Policies: The chosen eviction policy determines the order of eviction. Table 1 shows the supported policies and their scoring functions to get the next item for eviction. First, *LRU* orders by a normalized last access timestamp $T_a(o)/\theta$. Second, *DAG-Height* assumes that deep lineage traces have less reuse potential, and orders accordingly by $1/h(o)$. Third, similar to CO's [34] artifacts-materialization logic, *Cost&Size* aims to preserve objects with high computation costs $c(o)$ to size $s(o)$ ratio, scaled by $(r_h + r_m)$ (#accesses) to account for global reuse potential, and evicts the object with minimal $(r_h + r_m)\cdot c(o)/s(o)$. While *LRU* performs good in pipelines with temporal reuse locality, *DAG-Height* handles mini-batch scenarios better, where batches are sliced from the input dataset and reused across epochs. However, both *LRU* and *DAG-Height* make strong assumptions. In contrast, *Cost&Size*, together with disk spilling, performs well in a wide variety of scenarios as it tunes for global reuse utility (saved computation by size). Hence, *Cost&Size* is our default. We abandoned an Hybrid (weighted) strategy in favor of this parameter-free policy. Adaptive policies like ARC [66] that balance recency and utility is interesting future work.

4.4 Compiler Assistance

Lineage-based reuse at runtime level is valuable because many reuse opportunities are unknown during compilation due to conditional control flow. However, a pure runtime approach is not enough because some patterns are detected too late (after part of

these patterns are already evaluated). We address this dilemma by augmenting our runtime lineage cache with compiler assistance.

Unmarking Intermediates: In order to avoid cache pollution—and thus, unnecessary probing, and evictions—we added a program-level rewrite that unmarks intermediates for reuse. Unmarking disables probing and caching of a specific operation instance—even if its opcode is in the set of qualifying operations—if it is unlikely to be reused over the program lifetime. This rewrite has access to the entire control program and operation DAGs of a script, but performs unmarking conservatively because the lineage cache is used across script invocations through SystemDS' programmatic APIs. Examples are the computation of fully updated, local variables that depend recursively on previous loop iterations.

Reuse-aware Rewrites: We further added several reuse-aware DAG rewrites, which—if lineage-based reuse is enabled—prefer patterns that create additional reuse opportunities without hurting the runtime of normal plan execution (by cost estimates). There are several examples. First, for k-fold cross validation over `lm`, we construct the feature matrix from a list of folds $\mathbf{X} = \textbf{rbind}(\text{lfolds})$ and compute $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Applying, after function inlining, the partial rewrite $\textbf{dsyrk}(\textbf{rbind}(\mathbf{X}, \Delta\mathbf{X})) \rightarrow \textbf{dsyrk}(\mathbf{X}) + \textbf{dsyrk}(\Delta\mathbf{X})$ for $k-1$ folds during compilation allows us to avoid the repeated **rbind** operations, and compute and reuse the matrix multiplications just once per fold. Second, consider different feature projections via $\mathbf{R} = \mathbf{A}\,(\textbf{evect}[, 1:K])$ in PCA from Figure 5. If an outer loop calls PCA for different K, a dedicated rewrite speculatively computes $\mathbf{A}\,\textbf{evect}$ for more efficient partial reuse.

Reuse-aware Recompilation: SystemDS compiler marks a DAG for recompilation if sizes (dimensions, sparsity) of intermediates are unknown, and later adaptively recompiles plans during runtime at natural block boundaries. We extended this recompilation with further reuse-aware rewrites as it provides a great balance between reduced uncertainty of conditional control flow, and large-enough optimization scope. For example, in `stepLm` such rewrites allow partial reuse for $\textbf{dsyrk}(\textbf{cbind}(\mathbf{X}, \Delta\mathbf{X}))$, but also avoid the expensive materialization of $\textbf{cbind}(\mathbf{X}, \Delta\mathbf{X})$. Internally, the rewrites from Section 4.2 are shared by both, partial reuse and recompilation.

4.5 Limitations

Similar to Section 3.4, we summarize limitations, which we see as out-of-scope of the initial LIMA framework and thus, future work.

- *Unified Memory Management:* SystemDS has multiple memory managers such as the buffer pool for live variables, operation memory, and now the lineage cache. The static partitioning can cause unnecessary evictions, which makes a unified memory manager (as in Spark [76]) desirable.
- *Multi-level Partial Reuse:* Conceptually, we could also apply the idea of partial reuse—by composition—to blocks and functions. However, this would introduce significant complexity.
- *No Multi-location Caching:* Although we trace lineage for both local and distributed operations, the lineage cache currently only applies to local, in-memory objects.
- *Materialization of Lineage Cache:* Our reuse cache is designed for process-wide sharing, which also applies to collaborative notebook environments. Reusing across processes would require extensions for speculative materialization and cleanup.

5 EXPERIMENTS

Our experiments study the behavior of LIMA under various workloads regarding runtime overhead and reuse opportunities. We first conduct micro benchmarks to understand the performance of lineage tracing and cache probing, partial and multi-level reuse, and eviction policies. Subsequently, we explore the benefits of fine-grained, lineage-based reuse for end-to-end ML pipelines, on synthetic and real datasets, and in comparison with other ML systems. Overall, we observe low runtime overhead, and substantial reuse.

5.1 Experimental Setting

Setup: We ran all experiments on a Hadoop cluster with each node having a single AMD EPYC 7302 CPUs @ 3.0-3.3 GHz (16 physical/32 virtual cores), and 128 GB DDR4 RAM (peak performance is 768 GFLOP/s, 183.2 GB/s). The software stack comprises Ubuntu 20.04.1, Apache Hadoop 2.7, and Apache Spark 2.4. LIMA uses OpenJDK 1.8.0 with 110 GB max and initial JVM heap sizes.

Baselines: For the sake of a holistic evaluation, we compare LIMA with multiple baselines, including different SystemDS configurations, and other state-of-the-art approaches and systems:

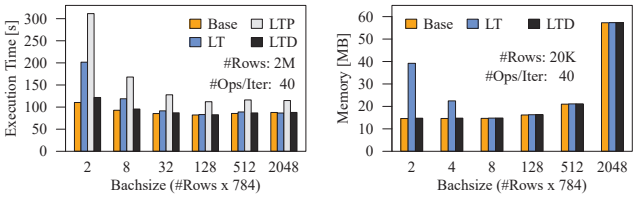
- *SystemDS:* Our main baseline is **Base** that refers to the default configuration of Apache SystemDS [15], but without any lineage tracing or lineage-based reuse. For lineage tracing and reuse in **LIMA**, we then use different configurations **LIMA-x**, introduced along the related experiments.
- *Coarse-grained:* Coarse-grained reuse in HELIX [103] and CO [34] uses dedicated DAG optimizers for reusing persistently materialized intermediates, and pruning unnecessary operations. For a fair comparison on equal footing (same runtime, best-case reuse), we hand-optimized the top-level ML pipelines at script level with reuse from memory.
- *ML Systems:* We compare with Scikit-learn [79] (**SKlearn**) and TensorFlow 2.3 [1] (**TF**), which are strong baselines regarding runtime and graph optimizations [59]. We use TF function annotations with AutoGraph [68] to compile composite ML pipelines into a single computation graph, which allows eliminating fine-grained redundancy as well.

Cache Configurations and Statistics: We expose and use various configurations, including different reuse types (full, partial, hybrid; and multi-level reuse), eviction policies (LRU, DAG-Height, Cost&Size), cache sizes, enabling disk spilling and compiler-assisted reuse. Additionally, LIMA collects various runtime statistics (e.g., cache misses, rewrite/spill times), which we report accordingly.

5.2 Micro Benchmarks

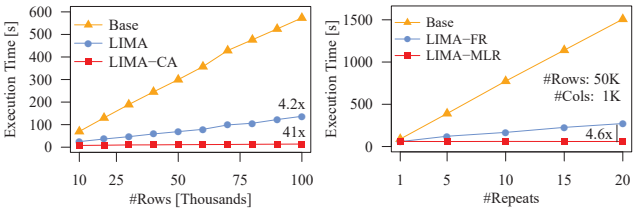
Before describing the end-to-end results, we conduct an ablation study of various LIMA aspects. These micro benchmarks focus on lineage tracing, cache probing, deduplication, partial rewrites, eviction policies, and multi-level reuse. We use simplified scripts, which are inspired by real workloads, but simple to understand.

Lineage Tracing: For understanding the overhead of lineage tracing, reuse probing, and deduplication, we explore a mini-batch scenario. We execute one epoch on a $2\text{M} \times 784$ matrix with different batch sizes b. Thus, we have $2\text{M}/b$ iterations, and every iteration contains 40 binary operations (ten times $\mathbf{X} = ((\mathbf{X} + \mathbf{X}) * i - \mathbf{X})/(i + 1)$). Figure 6 shows the results. First, in Figure 6(a), we see that



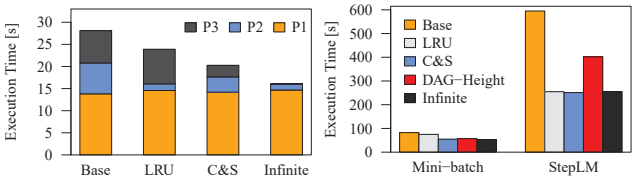
(a) Runtime Overhead (b) Space Overhead

Figure 6: Lineage Tracing Overhead (for one epoch).



(a) Partial Reuse (b) Multi-level Reuse

Figure 7: Partial Reuse and Multi-level Reuse.



(a) Pipeline with Phases (P1,P2,P3) (b) Pipeline Comparison

Figure 8: Cache Eviction Policies.

lineage tracing (**LT**), and lineage tracing, reuse probing (**LTP**) incurs substantial overhead for very small batch sizes ($b = 2$ and $b = 8$), but starting with $b = 32$ the overheads become moderate. In contrast, for lineage tracing with deduplication (**LTD**), the overheads become moderate even at $b = 2$ and negligible starting at $b = 8$. Base shows the best performance for $b = 128$ because per-operation overheads are amortized and until $b = 64$ intermediates still fit into L2 cache ($b \in [128, 2{,}048]$ fit in L3 cache, while for $b = 8{,}192$ there is an additional slowdown). Second, in Figure 6(b), we see similar characteristics for the space overhead of lineage tracing. Here, we use a reduced input matrix of 20K × 784 (as execution is substantially slower with forced garbage collection) and track[4] the maximum memory consumption after every instruction. For a batch size $b = 2$, we have 10K iterations and the lineage DAG of LT contains roughly 400K lineage items. The resulting space overhead compared to Base is about 24 MB (on average, 63 B per lineage item). Deduplication again significantly reduces the overhead to 10K dedup items (630 KB) in this scenario. The lineage cache adds a constant 5% space overhead relative to the heap size.

Partial Reuse: Furthermore, we evaluate partial reuse with a scenario inspired by `stepLm` [99]. We create a 100K × 500 matrix $\mathbf{X}$, another 100K × 1K matrix $\mathbf{Y}$, and compute $\mathbf{X}^\top\mathbf{X}$ once. In a for loop, we then execute 1,000 iterations of $\mathbf{Z}^\top\mathbf{Z}$ with $\mathbf{Z} = \textbf{cbind}(\mathbf{X}, \mathbf{Y}_i)$ and store a summary, which is the core of `stepLm`'s inner loop. Figure 7(a) shows the results of Base, LIMA, and LIMA with compiler assistance (**LIMA-CA**) for a varying number of rows. LIMA yields a 4.2x runtime improvement over Base by applying the partial rewrite **dsyrk**(**cbind**(X, ΔX)), which turns a compute-intensive **dsyrk** into reuse and an inexpensive matrix-vector multiplication for compensation. However, despite this partial rewrite, we still perform **cbind**(X, ΔX), which is expensive due to allocation and copy. LIMA-CA applies this rewrite during recompilation and thus, can eliminate the **cbind** for an improvement of 41x over Base.

Multi-level Reuse: Multi-level reuse eliminates redundancies at different hierarchy levels (e.g., functions, blocks, or operations) of a program, which helps reduce interpretation overhead and cache pollution. We conduct a micro benchmark of repetitive hyper-parameter optimization for iterative multi-class logistic regression with a 50K × 1K input matrix and 6 classes. We first call `MLogReg` with 40 different values of the regularization parameter λ. Then, we repeat the entire process 20 times. Figure 7(b) shows the runtime of Base, LIMA with full operation reuse (**LIMA-FR**), and LIMA with multi-level full reuse (**LIMA-MLR**). Both LIMA-FR and LIMA-MLR show good improvements, of 5.2x and 24.6x, respectively. MLR is 4.6x faster than FR because it avoids function interpretation overhead, whereas FR needs to retain all intermediates of the iterative computation cached, and use them one-by-one. Thus, MLR is less affected by evictions because Cost&Size is tuned to preserve group cache entries due to their higher computation time.

Eviction Policies: For evaluating eviction, we use ML pipelines with different reuse opportunities. The first pipeline has phases P1, P2, P3, where P1 is a loop of an expensive $\mathbf{X}\,\mathbf{Y}$ and **round**(X) with no reuse (which fills the cache), P2 is a nested loop with inexpensive additions $\mathbf{X} + i$ and reuse per outer iteration, and P3 is the same as P1—but with fewer iterations. Figure 8(a) shows a breakdown of execution time for Base, LRU, Cost&Size (C&S) and a hypothetical policy with unlimited cache. LRU fully reuses the intermediates of P2 by evicting P1 results, which leads to no reuse in P3. In contrast, C&S first evicts the $\mathbf{X}+i$ results, but due to cache misses, their score increase and they get reused. In P3, C&S reuses all matrix multiplies from P1. Second, Figure 8(b) compares the runtime of the mini-batch and StepLM pipelines. DAG-Height performs good on the mini-batch pipeline because it can reuse preprocessed batches, whereas with LRU, these batches are pushed out of cache during an epoch. On StepLM, we see a flipped characteristic, where incrementally added features lead to reuse potential on the end of large lineage DAGs and thus, LRU performs better. Due to accounting for cost, size, and cache references, C&S performs very good in both cases. Due to this robust behavior, C&S is our default policy.

5.3 ML Pipelines Performance

We now describe the performance impact of lineage-based reuse on end-to-end ML pipelines. For a balanced view of reuse opportunities, we evaluate a variety of pipelines with different characteristics.

Pipeline Summary: Table 2 summarizes the used ML pipelines and their parameters. These include grid search hyper-parameter optimization of (1) L2SVM (HL2SVM) and (2) linear regression (HLM); (3) cross-validated linear regression (HCV); (4) a weighted ensemble (ENS) of multi-class SVM (MSVM) and multi-class logistic regression (MLRG); and (5) a pipeline for dimensionality reduction using PCA as well as LM model training and evaluation (PCALM).

[4]In order to overcome measurement imprecision, we request JVM garbage collection (GC) until a dummy `WeakReference` has been collected for every measurement.

Table 2: Overview of ML Pipeline Use Cases.

Use Case	λ	icpt	tol	K/Wt	TP
HL2SVM	# = 70	{0,1}	10^{-12}	N/A	
HLM	$[10^{-5}, 10^{0}]$	{0, 1, 2}	$[10^{-12}, 10^{-8}]$	N/A	✓
HCV	$[10^{-5}, 10^{0}]$	{0, 1, 2}	$[10^{-12}, 10^{-8}]$	N/A	✓
ENS	# = 3	{1,2}	10^{-12}	[1K,5K]	(✓)
PCALM	N/A	N/A	N/A	$K \geq 10\%$	

Some of these pipelines leverage task-parallelism (TP in Table 2) as described in Section 3.3. For evaluating different data characteristics, we first use synthetic but later also real datasets. Theses ML pipelines are written as user-level scripts orchestrating SystemDS' built-in functions for pre-processing and ML algorithms.

Hyper-parameter Tuning (HL2SVM, HLM): Figures 9(a) and 9(b) show the runtime for HL2SVM and HLM. First, HL2SVM uses a 100K × 1K input matrix $\mathbf{X}$, calls L2SVM for 70 different λ values, each with and without intercept (i.e., bias), and uses the L2 norm to find the best parameters. We use an L_2-regularized linear SVM. Even though both outer and inner loop have no reuse opportunities, we see a nearly 2x improvement due to the reusable **cbind**$(\mathbf{X}, \mathbf{1})$ for intercept, initial loss, and gradient computations. Second, for HLM, we use the script from Example 1, and execute it for input matrices of varying number of rows [100K, 1M] and 100 columns. We see improvements of 2.6x and 12.4x (with and without task parallelism) for reasons explained in Example 1. With task parallelism, the reuse benefits are smaller because the `parfor` optimizer [19] reduces the parallelism of loop body operations, including $\mathbf{X}^\top\mathbf{X}$. With reuse, however, only a single thread executes $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$, whereas all other threads wait for the results. Together, the HL2SVM and HLM use cases show the spectrum of common reuse opportunities.

Cross Validation (HCV): HCV is similar to HLM but instead of LM, we use a cross-validated LM (with 16-fold, leave-one-out cross validation). We again compare Base and LIMA with and without task parallelism. Figure 9(c) shows the results, where we see improvements of 4x and 5.1x, respectively. Compared to HLM, we can no longer reuse $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ for different lambda parameters directly, but rely on partial rewrites to compute these operations once per fold and then assemble leave-one-out fold compositions. This characteristic, in turn, better utilizes task parallelism.

Ensemble Learning (ENS): The weighted ensemble learning pipeline has two phases. We train three multi-class SVM (MSVM) models—which internally leverage task parallelism—and three MLRG models as a weighted ensemble. Similar to L2SVM, MSVM and MLRG are also iterative with limited scope for reuse. The ensemble weights are then optimized via random search. Figure 9(d) shows the results for a 50K × 1K training dataset, 10K × 1K test dataset, 20 classes, and a varying number of [1K, 5K] weight configurations. We see again a solid 4.2x end-to-end improvement, which is due to reused $\mathbf{X}\mathbf{B}$ (of size **nrow**$(\mathbf{X})$ × #classes) matrix multiplication in the computation of weighted class probabilities.

Dimensionality Reduction (PCALM): Inspired by work on dimensionality reduction for downstream tasks [91], we use a PCA pipeline, PCALM—which enumerates different K, calls PCA to project K columns, LM and a predict function, and computes the adjusted-R^2. We vary K from 10% of all columns. Figure 9(e) shows that LIMA achieves up to a 5x improvement. Different calls to PCA

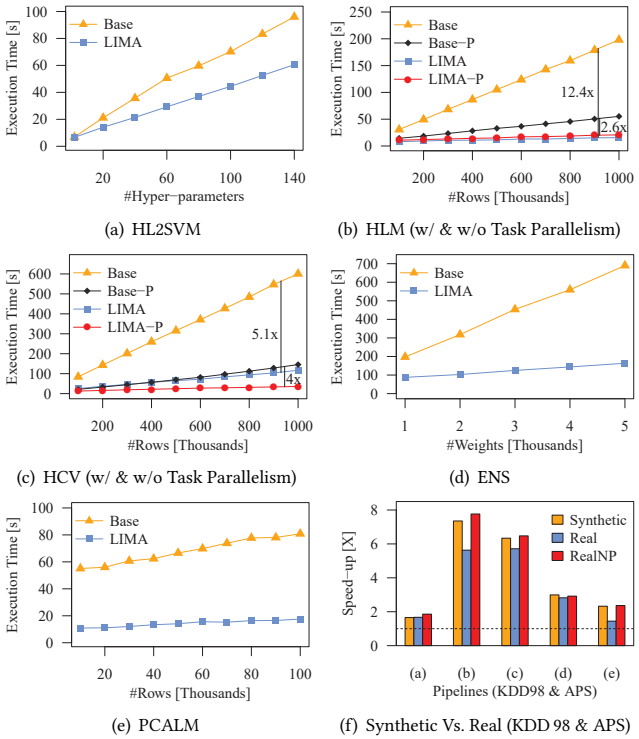


Figure 9: Performance of End-to-end ML Pipelines.

reuse the $\mathbf{A}^\top\mathbf{A}$ computation, the subsequent Eigen decomposition, and an overlapping matrix multiplication $\mathbf{A}$ **evects**$[, 1:K]$ (of size **nrow**$(\mathbf{A}) \times K$). Overlapping PCA outputs (projected features) further allow partial reuse in the following LM call, specifically $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. This PCA pipeline is another good example of significant fine-gained redundancy, even in modestly complex ML pipelines.

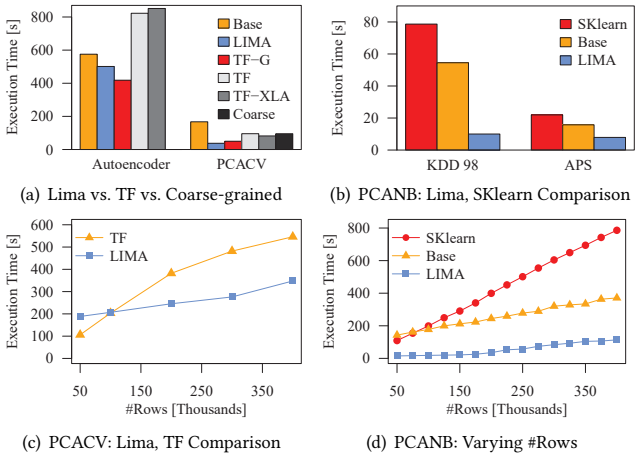
5.4 Real Datasets

Dataset Description: Lineage tracing and reuse are largely invariant to data skew. Besides synthetic data though, we also use real datasets from the UCI repository [38] to confirm the relative speedups. The real datasets are summarized in Table 3. APS is collected from various components of Scania Trucks for classifying failures of an Air Pressure System (APS). We pre-process this dataset by imputing missing values with mean and oversampling the minority class. The KDD 98 dataset is a regression problem for the return from donation campaigns. For pre-processing, we recoded categorical, binned continuous (10 equi-width bins), and one-hot encoded both binned and recoded features. Column 4 and 5 of Table 3 show the data dimensions after pre-processing.

Reuse Results: Figure 9(f) compares the speedups obtained from synthetic and real datasets with and without pre-processing (Real & RealNP). The baseline synthetic datasets are generated to

Table 3: Dataset Characteristics.

Dataset	nrow($\mathbf{X}_0$)	ncol($\mathbf{X}_0$)	nrow($\mathbf{X}$)	ncol($\mathbf{X}$)	ML Alg.
APS	60,000	170	70,000	170	2-Class
KDD 98	95,412	469	95,412	7,909	Reg.



(a) Lima vs. TF vs. Coarse-grained (b) PCANB: Lima, SKlearn Comparison

(c) PCACV: Lima, TF Comparison (d) PCANB: Varying #Rows

Figure 10: ML Systems Comparison.

match the data characteristics of the real datasets. Scenarios (a), (b), (c) and (e) show similar speedups for L2SVM, HLM, HCV, and PCALM with real data using the KDD 98 dataset. For L2SVM (a), we converted the target column to 2-class label, and for PCALM (e), we skipped one-hot encoding to reduce the influence of Eigen decomposition. Scenario (d) shows a similar result for ENS on the APS dataset. These experiments validate that lineage-based reuse is largely independent of data skew of real datasets.

5.5 ML Systems Comparison

So far, we compared SystemDS and LIMA in an equivalent compiler and runtime infrastructure. Additionally, we also compare with (1) coarse-grained reuse, (2) global graph construction and CSE in TensorFlow (TF) [1, 68], and (3) Scikit-learn (SKlearn) [79] as a state-of-the-art ML library. We use three new pipelines for comparison.

Autoencoder: For comparing TF on mini-batch, NN algorithms, we use an Autoencoder with two hidden layers of sizes 500 and 2 (four weight matrices), and varying batch size. For avoiding transforming the entire input, we build a feature-wise pre-processing map including normalization, binning, recoding, and one-hot encoding, and then apply this map (as a Keras pre-processing layer) batch-wise in each iteration. SystemDS is ran with code generation (i.e., operator fusion in Section 3.3) for both Base and LIMA. Figure 10(a) shows the results on the KDD 98 dataset for a batch size 256 and 10 epochs. Even though Autoencoder has no reuse opportunities, LIMA shows a 15% improvement over Base by reusing the batch-wise pre-processing. TF—a specialized system for mini-batch training—performs slightly better than LIMA in graph mode (TF-G), whereas eager mode (TF) and XLA code generation for CPU (TF-XLA) are substantially slower. In additional experiments, we found higher SystemDS overhead for small batch sizes, but converging performance with batch sizes beyond 2,048.

PCA and Cross Validation (PCACV): As a pipeline for evaluating reuse, we apply PCA and cross validation in two phases. The first phase varies K for PCA, and the second varies the regularization parameter λ for LM with cross validation (32 folds, leave-one-out), and evaluates the adjusted-R^2. PCACV is then compared with TF and coarse-grained reuse. For a fair and interpretable comparison, we disable both task parallelism in SystemDS and inter-operator parallelism in TF. Figure 10(a)-right shows the results for PCACV on the KDD 98 dataset. The coarse-grained reuse shows improvements over Base (by reusing the PCA result), but is limited to top-level redundancies. In contrast, both LIMA and TF-G eliminate fine-grained redundancies (via CSE in TF-G) but LIMA yields a 25% runtime improvement over TF-G due to partial reuse across folds. Figure 10(c) varies the number of rows $\in$ [50K, 400K], where we see an improvement up to 2x by LIMA over TF. For larger datasets, TF ran out-of-memory, likely because the global graph misses eviction mechanisms for reused intermediates.

PCA and Naïve Bayes (PCANB): In addition, we use a PCA and Naïve Bayes (NB) pipeline, again with two phases: varying K for PCA, and hyper-parameter tuning for NB. Due to minor algorithmic differences, we tune Laplace smoothing in LIMA, but feature variance (var_smoothing) in SKlearn. Figure 10(b) shows the results for PCANB on the KDD 98 and APS datasets. LIMA performs 8x and 2.8x better than SKlearn for the KDD 98 and APS datasets, respectively. LIMA reuses again full and partial PCA intermediates, as well as partial intermediates of the main `aggregate` operation in different calls to NB. Figure 10(d) shows the runtime with varying rows $\in$ [50K, 400K], 1K columns and 20 classes. LIMA is up to 10x faster than SKlearn. Due to differences in implementation of PCA (SVD vs. Eigen), SKlearn is faster for smaller data sizes, whereas Base shows better scalability with increasing data size.

Conclusion: Overall, SystemDS with LIMA shows competitive performance, leveraging rewrite and reuse opportunities not yet exploited in other systems like TensorFlow and Scikit-learn.

6 RELATED WORK

Our fine-grained, multi-level operation lineage tracing and reuse is related to traditional data provenance, model management, reuse of query intermediates and intermediates in ML pipelines, as well as the incremental maintenance of intermediates in ML systems.

Data Provenance: Data provenance for tracking the origin and creation of data has been extensively studied in the data management literature [27, 42, 92]. Common types of data provenance are (1) why-provenance [21, 31] (via input tuple "witnesses"), (2) how-provenance [44, 45] (via provenance polynomials), and (3) why-not-provenance (why eliminated) [23]. From an execution perspective, we distinguish eager and lazy provenance, which obtain lineage during execution or on demand [27, 80]. Furthermore, there is also work on fine-grained, tuple-oriented data provenance in data flow programs such as MapReduce [33], Spark [105], or PigLatin [74]. Examples are RAMP [50], HadoopProv [4], and Newt [63] for MapReduce, Lipstick [5] for PigLatin, and Titian [51] for Spark. Similar to LIMA, the Lipstick system [5] uses a lineage graph for fine-grained provenance in a hierarchy of Pig Latin modules, but unlike LIMA, Lipstick operates at tuple granularity and does not deduplicate repeated structures. Provenance management in scientific workflows introduced additional ideas on user-centric provenance information [6, 32, 64], and caching [10, 22]. Recent work focuses on fine-grained provenance for linear algebra via provenance polynomials on matrix partitions [104], efficient provenance tracking via RID-based indexes and query-aware optimizations [80], provenance for ETL workflows and entity resolution

via how-provenance [110], provenance for data citation [102] and natural language claims [109], and provenance for blockchains via Merkle DAGs [82]. In contrast to tuple-oriented provenance, we perform logical lineage tracing of linear algebra operations, operate at multiple levels of conditional control flow, and primarily focus on efficient lineage tracing for both reproducibility and reuse.

Dataset and Model Management: Recent work on catalogs for dataset and model management includes Google Goods [46], SAP Data Hub [47], and DataHub [13], but also data market platforms [41], which all store provenance information to track the origins and preparation of datasets. In the context of open science, similar data catalogs are established under the FAIR data principles [101], where principle R1.2 requires that "(meta)data are associated with detailed provenance". Related projects like Apache Atlas [9]—as used in Microsoft's Enterprise ML vision [2]—offers APIs for storing provenance form different systems. DataHub [13] and DEX [24] further provided git-like dataset versioning with delta encoding; MISTIQUE [97] extended this line of work by lossy deduplication, compression, and adaptive materialization. Further related work includes ML model versioning, experiment tracking, and model management. For example, van der Weide et al. manually version ML pipeline functions, manage their pipeline dependencies, and reuse intermediates [96]. TensorFlow [1] allows programmatic tracking of variables (e.g., loss) for experiment visualization in TensorBoard, while MLflow [25, 106] provides APIs for tracking model parameters and resulting accuracy. Similarly, the ML library Tribuo [93] attaches provenance information to models and datasets for reproducibility. More specialized model management—like ModelHub [67] and ModelDB [98]—further provides model versioning and means of querying these model versions. In contrast to such coarse-grained versioning, we provide means of fine-grained lineage tracing of linear algebra programs *inside* ML systems.

Reuse of Query Intermediates: There is also a long history on reusing work in database systems. The spectrum ranges from buffer pool page caching, over scan sharing [7, 94], request batching [60], and adaptive indexing/cracking [49], to multi-query optimization [81] and materialized views [3, 55]. Our work has been inspired by the seminal work on recycling intermediates in MonetDB [52, 53] and transient materialized views [111]. Both of these approaches leverage intermediates that are anyway materialized in memory (or materialized via `spool` operators) for future reuse. Accordingly, several aspects like runtime integration and cache eviction policies of our approach share similarities. However, in contrast to existing work, we provide means of efficient lineage tracing and reuse for *linear algebra programs*, which entails dedicated loop deduplication strategies, multi-level lineage in conditional control flow, as well as linear-algebra-specific rewrites for full and partial reuse.

Reuse of ML Pipeline Intermediates: Exploratory data science workflows also have large reuse opportunities. Already notebooks—which preserve the state of cells—and extensions for dataset discovering [108] can be viewed as a form of manual reuse. Recent work leverages lineage tracing for notebook state-safety inspection [20, 65]. For example, NBSAFETY [65] provides a custom Jupyter kernel that highlights unsafe cell executions (via static analysis and runtime lineage tracing), but does not trace fine-grained lineage of library function calls. Early work on automated reuse was then introduced for optimizing ML pipelines in Columbus [107] and KeystoneML [89], which both reason about materialization and reuse for exact and approximate reuse, within and across pipelines. ML serving systems also apply means of reuse. Examples are function result caching in Clipper [29] (e.g., caching frequently translated words), CSE across prediction programs in PRETZEL [61], and short-circuiting via reference labels in NoScope [56]. More recent work include Alpine Meadow [86], HELIX [103], and VAMSA [71], which include operations for data preparation, ML training, and model evaluation. Similar to our compiler-assisted reuse and cache eviction policies, HELIX uses a cost model of load, materialization and computation costs, as well as a plan selection heuristic. Most of these systems rely on existing ML systems like Scikit-learn [79] and reason about the pipeline DAG, which is a coarse-grained, top-level view of ML pipelines. In contrast, we exploit fine-grained full and partial reuse at multiple control flow granularities.

Incremental Maintenance of ML Models: Partial reuse of intermediates is closely related to the incremental maintenance of ML models and intermediates. Model serving systems like Velox [28] and classification Views in Hazy [57] apply online learning for model adaptation. In addition, there is also work on exact incremental maintenance—in linear algebra programs and related abstractions—such as LINVIEW [72], F-IVM [73], and MauveDB [35]. An often exploited opportunity is maintaining $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ and $\mathbf{b} = \mathbf{X}^\top\mathbf{y}$, via $\mathbf{A}' = \mathbf{A} + \Delta\mathbf{X}^\top\Delta\mathbf{X}$ and $\mathbf{b}' = \mathbf{b} + \Delta\mathbf{X}^\top\Delta\mathbf{y}$. Recent work also leveraged this property for decremental updates [84] (e.g., to remove tuples for GDPR regulations). Further work includes incremental grounding and inference in DeepDive [88], and incremental computation of occlusion-based explanations for CNNs [69, 70]. In contrast to the incremental maintenance of intermediates, our partial reuse is more general because it allows rewrites to augment intermediates by complex compensation plans.

7 CONCLUSIONS

To summarize, we introduced LIMA, a framework for fine-grained lineage tracing and reuse in ML systems. Multi-level lineage tracing for functions, blocks, and operations—with deduplication for loops and functions—reduces the overhead of lineage tracing and reuse, and seamlessly supports fused operators. Compiler-assisted, full and partial reuse during runtime allows removing coarse- and fine-grained redundancy at the different levels of hierarchically composed ML pipelines. Even for modestly-sized ML pipelines, our experiments have shown robust improvements across a variety of workloads. In conclusion, as the complexity of ML pipelines increases both horizontally (additional sub tasks), and vertically (additional hierarchy levels), increasing redundancy is inevitable and difficult to address by library developers or users. Conditional control flow further renders global operator graphs and common subexpression elimination during compilation ineffective. In contrast, a compiler-assisted runtime-based lineage cache proved effective to overcome these challenges. Despite a dependency on system internals, the same concepts are broadly applicable in many modern ML systems. Interesting future work includes (1) the combination with persistent materialization [97, 103], especially in multi-tenant and federated environments, (2) multi-location caching for local, distributed, and multi-device settings, as well as (3) extended lineage support for model debugging and fairness constraints [43].

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.

[2] Ashvin Agrawal et al. 2020. Cloudy with High Chance of DBMS: A 10-year Prediction for Enterprise-Grade ML. In *CIDR*.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505.

[4] Sherif Akoush, Ripduman Sohan, and Andy Hopper. 2013. HadoopProv: Towards Provenance as a First Class Citizen in MapReduce. In *TaPP*.

[5] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB* 5, 4 (2011), 346–357.

[6] Manish Kumar Anand, Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. 2009. Exploring Scientific Workflow Provenance Using Hybrid Queries over Nested Data and Lineage Graphs. In *SSDBM*. 237–254.

[7] Subi Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Leopoldo Perez. 2010. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD*. 519–530.

[8] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*. 173–182.

[9] Apache Atlas. 2020. Open Metadata Management and Governance. https://atlas.apache.org/.

[10] Louis Bavoil, Steven P. Callahan, Carlos Eduardo Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. 2005. VisTrails: Enabling Interactive Multiple-View Visualizations. In *IEEE Vis*. 135–142.

[11] Denis Baylor et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *SIGKDD*. 1387–1395.

[12] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98.

[13] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR*.

[14] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.

[15] Matthias Boehm et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*.

[16] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62.

[17] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems*. Morgan & Claypool Publishers.

[18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768.

[19] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564.

[20] Mike Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *CIDR*.

[21] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*, Vol. 1973. 316–330.

[22] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization meets Data Management. In *SIGMOD*. 745–747.

[23] Adriane Chapman and H. V. Jagadish. 2009. Why Not?. In *SIGMOD*. 523–534.

[24] Amit Chavan and Amol Deshpande. 2017. DEX: Query Execution in a Delta-based Storage System. In *SIGMOD*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). 171–186.

[25] Andrew Chen et al. 2020. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *SIGMOD Workshop DEEM*. 5:1–5:4.

[26] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. 578–594.

[27] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* 1, 4 (2009), 379–474.

[28] Daniel Crankshaw, Peter Bailis, Joseph E. Gonzalez, Haoyuan Li, Zhao Zhang, Michael J. Franklin, Ali Ghodsi, and Michael I. Jordan. 2015. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. In *CIDR*.

[29] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. 613–627.

[30] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* 8, 12 (2015), 1466–1477.

[31] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.* 25, 2 (2000), 179–227.

[32] Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. 2007. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.* 30, 4 (2007), 44–50.

[33] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[34] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*. 1701–1716.

[35] Amol Deshpande and Samuel Madden. 2006. MauveDB: Supporting Model-based User Views in Database Systems. In *SIGMOD*. 73–84.

[36] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah A. Smith. 2020. Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping. *CoRR* abs/2002.06305 (2020).

[37] Xin Luna Dong and Theodoros Rekatsinas. 2018. Data Integration and Machine Learning: A Natural Synergy. In *SIGMOD*. 1645–1650.

[38] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[39] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*.

[40] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph using Graph Substitutions. *PVLDB* 13, 11 (2020), 2734–2746.

[41] Raul Castro Fernandez, Pranav Subramaniam, and Michael J. Franklin. 2020. Data Market Platforms: Trading Data Assets to Solve Data Problems. *PVLDB* 13, 11 (2020), 1933–1947.

[42] Boris Glavic and Klaus R. Dittrich. 2007. Data Provenance: A Categorization of Existing Approaches. In *BTW*. 227–241.

[43] Stefan Grafberger, Julia Stoyanovich, and Sebastian Schelter. 2021. Lightweight Inspection of Data Preprocessing in Native Machine Learning Pipelines. In *CIDR*.

[44] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2007. Update Exchange with Mappings and Provenance. In *VLDB*. 675–686.

[45] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *PODS*. ACM.

[46] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *SIGMOD*. 795–806.

[47] Marc Hartz. 2020. SAP Data Intelligence: Next evolution of SAP Data Hub. https://blogs.sap.com/2020/03/20/sap-data-intelligence-next-evolution-of-sap-data-hub/.

[48] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*. 1–12.

[49] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78.

[50] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *CIDR*. 273–283.

[51] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *PVLDB* 9, 3 (2015), 216–227.

[52] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD*. 309–320.

[53] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2010. An Architecture for Recycling Intermediates in a Column-store. *ACM Trans. Database Syst.* 35, 4 (2010), 24:1–24:43.

[54] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *SOSP*. 47–62.

[55] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB* 11, 7 (2018), 800–812.

[56] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB* 10, 11 (2017), 1586–1597.

[57] Mehmet Levent Koc and Christopher Ré. 2011. Incrementally Maintaining Classification using an RDBMS. *PVLDB* 4, 5 (2011), 302–313.

[58] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.

[59] Rasmus Munk Larsen and Tatiana Shpeisman. 2019. TensorFlow Graph Optimizations. Stanford guest lecture, https://web.stanford.edu/class/cs245/slides/

TFGraphOptimizationsStanford.pdf.
[60] Rubao Lee, Minghong Zhou, and Huaming Liao. 2007. Request Window: an Approach to Improve Throughput of RDBMS-based Data Integration System by Utilizing Data Sharing Across Concurrent Distributed Queries. In *VLDB*. 1219–1230.
[61] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *OSDI*. 611–626.
[62] Edo Liberty et al. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. 731–737.
[63] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable Lineage Capture for Debugging DISC Analytics. In *SOCC*. 17:1–17:15.
[64] Bertram Ludäscher et al. 2006. Scientific Workflow Management and the KEPLER System. *Concurr. Comput. Pract. Exp.* 18, 10 (2006), 1039–1065.
[65] Stephen Macke, Hongpu Gong, Doris Jung Lin Lee, Andrew Head, Doris Xin, and Aditya G. Parameswaran. 2021. Fine-Grained Lineage for Safer Notebook Interactions. *PVLDB* 14, 06 (2021), 1093–1101.
[66] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*.
[67] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In *ICDE*. 1393–1394.
[68] Dan Moldovan et al. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. *SysML* (2019).
[69] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations. In *SIGMOD*. 1589–1606.
[70] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2020. Query Optimization for Faster Deep CNN Explanations. *SIGMOD Rec.* 49, 1 (2020), 61–68.
[71] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *SIGKDD*. 1542–1551.
[72] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. 2014. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *SIGMOD*. 253–264.
[73] Milos Nikolic and Dan Olteanu. 2018. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*. 365–380.
[74] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*. 1099–1110.
[75] Hong Ooi and Stephen Weston. 2019. *doSNOW: Foreach Parallel Adaptor for the snow Package*. https://cran.r-project.org/web/packages/doSNOW/doSNOW.pdf.
[76] Andrew Or and Josh Rosen. 2015. Unified Memory Management in Spark 1.6. https://issues.apache.org/jira/secure/attachment/12765646/unified-memory-management-spark-10000.pdf.
[77] Shoumik Palkar et al. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015.
[78] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 8024–8035.
[79] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12 (2011), 2825–2830.
[80] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained Lineage at Interactive Speed. *PVLDB* 11, 6 (2018), 719–732.
[81] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*. 249–260.
[82] Pingcheng Ruan, Gang Chen, Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-Grained, Secure and Efficient Data Provenance for Blockchain. *PVLDB* 12, 9 (2019), 975–988.
[83] Amit Sabne. 2020. XLA: Compiling Machine Learning for Peak Performance. SIGMOD Workshop DEEM, Industry Keynote.
[84] Sebastian Schelter. 2020. "Amnesia" - Machine Learning Models That Can Forget User Data Very Fast. In *CIDR*.
[85] Sebastian Schelter, Andrew Palumbo, Shannon Quinn, Suneel Marthi, and Andrew Musselman. 2016. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. *NIPS MLSys* (2016).
[86] Zeyuan Shang et al. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*. 1171–1188.
[87] Gaurav Sharma and Jos Martin. 2009. MATLAB: A Language for Parallel Computing. *Int. J. Parallel Program.* 37, 1 (2009), 3–36.
[88] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental Knowledge Base Construction Using DeepDive. *PVLDB* 8, 11 (2015), 1310–1321.
[89] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*. 535–546.
[90] Arvind K. Sujeeth et al. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*. 609–616.
[91] Sahaana Suri and Peter Bailis. 2019. DROP: A Workload-Aware Optimizer for Dimensionality Reduction. In *SIGMOD Workshop DEEM*. 1:1–1:10.
[92] Wang Chiew Tan. 2007. Provenance in Databases: Past, Current, and Future. *IEEE Data Eng. Bull.* 30, 4 (2007), 3–12.
[93] Tribuo. 2021. Machine Learning in Java. https://tribuo.org/.
[94] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2009. Predictable Performance for Unpredictable Workloads. *PVLDB* 2, 1 (2009), 706–717.
[95] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Comp.S&E* 13, 2 (2011).
[96] Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. 2017. Versioning for End-to-End Machine Learning Pipelines. In *SIGMOD Workshop DEEM*. 2:1–2:9.
[97] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). 1285–1300.
[98] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.* 41, 4 (2018), 16–25.
[99] William N. Venables and Brian D. Ripley. 2002. *Modern Applied Statistics with S, 4th Ed.* Springer.
[100] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *PVLDB* 13, 11 (2020), 1919–1932.
[101] Mark D. Wilkinson et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data* 3, 1 (2016).
[102] Yinjun Wu, Abdussalam Alawini, Daniel Deutch, Tova Milo, and Susan B. Davidson. 2019. ProvCite: Provenance-based Data Citation. *PVLDB* 12, 7 (2019), 738–751.
[103] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460.
[104] Zhepeng Yan, Val Tannen, and Zachary G. Ives. 2016. Fine-grained Provenance for Linear Algebra Operators. In *TaPP*, Sarah Cohen Boulakia (Ed.).
[105] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.
[106] Matei Zaharia et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.
[107] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*. 265–276.
[108] Yi Zhang and Zachary G. Ives. 2019. Juneau: Data Lake Management for Jupyter. *PVLDB* 12, 12 (2019), 1902–1905.
[109] Yi Zhang, Zachary G. Ives, and Dan Roth. 2020. "Who said it, and Why?" Provenance for Natural Language Claims. In *ACL*. 4416–4426.
[110] Nan Zheng, Abdussalam Alawini, and Zachary G. Ives. 2019. Fine-Grained Provenance for Matching & ETL. In *ICDE*. 184–195.
[111] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD*. 533–544.