SystemML's Optimizer:
Plan Generation for Large-Scale Machine Learning Programs

Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald,
Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian

IBM Research – Almaden, San Jose, CA, USA

Abstract

SystemML enables declarative, large-scale machine learning (ML) via a high-level language with R-like syntax. Data scientists use this language to express their ML algorithms with full flexibility but without the need to hand-tune distributed runtime execution plans and system configurations. These ML programs are dynamically compiled and optimized based on data and cluster characteristics using rule- and cost-based optimization techniques. The compiler automatically generates hybrid runtime execution plans ranging from in-memory, single node execution to distributed MapReduce (MR) computation and data access. This paper describes the SystemML optimizer, its compilation chain, and selected optimization phases for generating efficient execution plans.

1 Introduction

Large-scale machine learning has become critical to the success of many business applications such as customer experience analysis, log analysis, social data analysis, churn analysis, cyber security, and many others. Both, ever increasing data sizes and analysis complexity naturally lead to large-scale, data and task parallel ML.

SystemML [2, 5, 11] aims at declarative, large-scale ML via a compiler-based approach. ML programs such as regression, classification, or clustering are expressed in a high-level language with R-like syntax called DML (Declarative Machine learning Language). SystemML compiles these programs, applies rewrite rules and cost-based optimizations according to data and cluster characteristics, and generates runtime execution plans. Runtime plans may include in-memory, single node computations and parallel computations on MapReduce clusters. The high-level language significantly increases the productivity of data scientists compared to low-level programming in MapReduce because it provides full flexibility and data independence as well as efficiency and scalability via automatic optimization. The compiler-based approach of SystemML differs from existing work on large-scale ML libraries like Mahout [10], MADlib [6] or MLlib [9], which mostly provide fixed algorithms and runtime plans and often expose physical data representations. However, there is also recent work like Cumulon [7] or future plans of Mahout [4] that follow SystemML towards declarative, large-scale ML.

Our DML language allows to express a rich set of ML algorithms ranging from descriptive statistics, classification, clustering, regression, to matrix factorization. To demonstrate the complexity of compiling arbitrary ML programs, we discuss two examples of regression and classification algorithms, and highlight their core computations. Optimizing these computations is key to algorithm performance and scalability.

Copyright 2014 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Example 1 (Linear Regression (Normal Equations) using $\mathbf{X}^\top\mathbf{X}$): Linear regression predicts a dependent continuous variable $\mathbf{y}$ from multiple continuous feature variables $\mathbf{X}$ by finding coefficients $\beta_0, \beta_1, \ldots, \beta_m$ using a regularization constant $\lambda > 0$. This formulation has the following matrix linear equation form: $\mathbf{A}\beta = \mathbf{X}^\top\mathbf{y}$, where $\mathbf{A} = \mathbf{X}^\top\mathbf{X} + \text{diag}(\lambda)$. In SystemML's DML syntax, this computation is expressed as:

```
1:   A = t(X) %*% X + diag(lambda);
2:   b = t(X) %*% y;
3:   beta = solve(A, b);
```

where `solve` is a built-in function for solving a linear system of equations that takes a small matrix $\mathbf{A}$ and a vector $\mathbf{b}$, and returns the coefficient vector β. Regarding data sets with many observations, i.e. rows, computing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are by far the most expensive operations. First, for $\mathbf{X}^\top\mathbf{X}$, a dedicated implementation method could be applied which exploits the unary input characteristic avoiding data shuffling, and leveraging local partial aggregation, as well as the result symmetry by doing only half the computation. Second, for $\mathbf{X}^\top\mathbf{y}$, assuming that vector $\mathbf{y}$ fits in memory, we can distribute $\mathbf{y}$ via Hadoop distributed cache and again avoid data shuffling and leverage local partial aggregation. We can also rewrite $\mathbf{X}^\top\mathbf{y}$ to $(\mathbf{y}^\top\mathbf{X})^\top$ in order to prevent the transpose of $\mathbf{X}$ altogether. In this example, all operations can be packed into a single MR job that reads $\mathbf{X}$ only once.

Example 2 (Logistic Regression via Trust Region Method): Our second running example is logistic regression that learns a binary classifier $h : \Re^m \rightarrow \{\pm 1\}$ where h is expressed in terms of a linear function $h(x) = 1$ if $\beta^\top x \geq 0$ and $h(x) = -1$ otherwise. The goal is then to learn the model parameters β. Given a fully labeled training set $\{(x_i, y_i)\}$, where x_i denotes the ith example and $y_i \in \{\pm 1\}$ its label, one can learn β with regularization constant λ by minimizing the following objective function: $\lambda/2\beta^\top\beta + \sum_i \log\left[1 + \exp(y_i \beta^\top x_i)\right]$. There are numerous ways to perform the above optimization. The trust-region Newton method works well in high-dimensional feature spaces [8] and can be expressed in DML as follows (simplified):

```
 1:   while( sum(g^2) > exit_g2 & i < max_i ) {
 2:      sb = zeros_nf;  r = g;  r2 = sum(r^2);  exit_r2 = 0.01 * r2;
 3:      d = -r;  tb_reached = FALSE; j = 0;
 4:      while( r2 > exit_r2 & (! tb_reached) & j < max_j ) {
 5:         Hd = lambda * d + t(X) %*% diag(v) %*% X %*% d;    # core computation
 6:         a = r2 / sum(d * Hd);
 7:         [a,tb_reached] = ensure_tb(a,sum(d^2),2*sum(sb*d),sum(sb^2)-delta^2);
 8:         sb = sb + a * d;  r = r + a * Hd;  r2_new = sum(r^2);
 9:         d = - r + (r2_new / r2) * d;  r2 = r2_new; j = j + 1;  }
10:      p = 1.0 / (1.0 + exp(-y * (X %*% (b + sb))));
11:      so = -sum(log (p)) + 0.5 * lambda * sum((b + sb)^2) - o; i = i + 1;
12:      delta = update_tr(delta,sqrt(sum(sb^2)),so,sum(sb*g),0.5*sum(sb*(r+g)));
13:      if( so < 0 ) {
14:         b = b + sb;  o = o + so;  v = p * (1 - p);
15:         g = - t(X) %*% ((1 - p) * y) + lambda * b;  }
16:   }
```

In the above ML program, we learn a logistic regression classifier using two nested loops. This template of two nested loops is similar to many implementations of unconstrained optimization problems. The inner `while` loop uses an iterative solver to compute the optimal update to model parameters β. Note that the subroutines `ensure_tb` and `update_tr` ensure trust bounds and update the trust region but are non-essential from a computational perspective. The core computation of the inner loop is $\mathbf{X}^\top(diag(\mathbf{v})\mathbf{X}\mathbf{d})$ (line 5). Analyzing the data flow, we can exploit important characteristics. First, since $\mathbf{X}\mathbf{d}$ produces a column vector, we can rewrite the expression to $\mathbf{X}^\top(\mathbf{v} \cdot \mathbf{X}\mathbf{d})$, eliminating the `diag` operation and replacing the subsequent matrix multiplication with a cell-wise vector multiplication of $\mathbf{v} \cdot (\mathbf{X}\mathbf{d})$. Second, since a block in $\mathbf{X}$ is equivalent to the related block in $\mathbf{X}^\top$, we can—if $\mathbf{v}$ and $\mathbf{d}$ fit in memory—invoke a dedicated operator that reads $\mathbf{v}$ and $\mathbf{d}$ through Hadoop distributed cache and does a single pass over $\mathbf{X}$ to compute the entire matrix multiplication chain.

In general, SystemML parses and compiles these ML programs to hybrid runtime execution plans. Runtime operations range from in-memory, single node operations to large-scale cluster operations via MapReduce jobs. Hybrid execution plans enable us to scale up and down as required by input data and program characteristics.

System Architecture: The overall SystemML architecture has different layers. The *language* contains a rich set of statistical functions, linear algebra operations, control structures, user-defined and external functions, recursion, and ML-specific operations. The parser produces directed acyclic graphs (DAGs) of *high-level operators* (HOPs) per block of statements as defined by control structures. HOPs such as matrix multiplication, unary and binary operations, or reorg operations operate on intermediates of matrices and scalars. Various optimizations are applied on HOP DAGs, including operator ordering and selection. Operator selection is significant as—similar to relational query optimization—the runtime supports for expensive operations such as matrix multiplication, several alternative physical operators, which are chosen depending on data and cluster characteristics. HOP DAGs are transformed to *low-level operator* (LOP) DAGs. Low-level operators such as grouping, aggregate, transform, or binary operations operate on runtime-specific intermediates such as key-value pairs in MapReduce. Given assigned LOP execution types, the LOPs of a DAG are piggybacked into a workflow of MR jobs in order to minimize data scans, for operator pipelining, and latency reduction. LOPs have equivalent *runtime* implementations, i.e., runtime instructions which are either executed in-memory of the single node control program (CP) or on MapReduce (MR). MR instructions are executed via few generic MR job types.

Discussion SystemML Design: The overall goal of SystemML is to allow declarative ML for a wide variety of use cases, where ML algorithms can be implemented independent of input data and cluster characteristics. Four major requirements influenced the design of SystemML. First, the need for *full flexibility* in specifying new ML algorithms and customizing existing algorithms led to a domain-specific language for ML. A library interface of ML algorithms could not achieve this level of flexibility. Second, the choice of operations in our DML language was directly influenced by the goal of *data independence*. Semantic operations such as linear algebra operations make us independent from the underlying physical representation such as dense/sparse representations, row/column major layout, or blocking configurations. In contrast to general purpose parallel programming languages, the DML approach simplifies optimization because the operation semantics are preserved and with that they are easier to reason about. The approach also simplifies the runtime as alternative physical operator implementations may be provided. Third, the requirement of running the same ML algorithms on very small to very large data sets combined with the need for *efficiency and scalability* demands automatic optimization and hybrid runtime plans of in-memory single node operations, and large-scale cluster computations. Fourth, our current view on declarative ML is focused on *specified algorithm semantics* and execution plan generation for performance and scalability only. We do not optimize for accuracy because this would contradict the language flexibility and would make it harder to understand, debug, and control the algorithm behavior.

2 Compilation Chain Overview

SystemML uses a well defined compilation chain in order to generate an executable runtime program (execution plan) for a given DML script. Figure 1 shows an overview of this chain and associates the individual compilation and optimization phases to SystemML's general architecture, where the example DML expression on the right refers back to Example 2 (Logistic Regression, line 10). Our overall optimization objective is to minimize the script execution time under hard memory constraints given a certain cluster configuration.

Language-Level: First, we start the compilation process by *parsing* the given DML script into a hierarchical representation of statement blocks and statements, where statement blocks are defined by the program structure in terms of control flow constructs like branches, loops, or calls to user defined functions. This parsing step is responsible for lexical and syntactic analysis but also for aspects like basic order of operations (e.g., multiply/plus). In detail, we use a parser generator to create this parser based on our specific DML grammar. Second, we do a classic *live variable analysis* [1] in terms of a data flow analysis over statement blocks. We

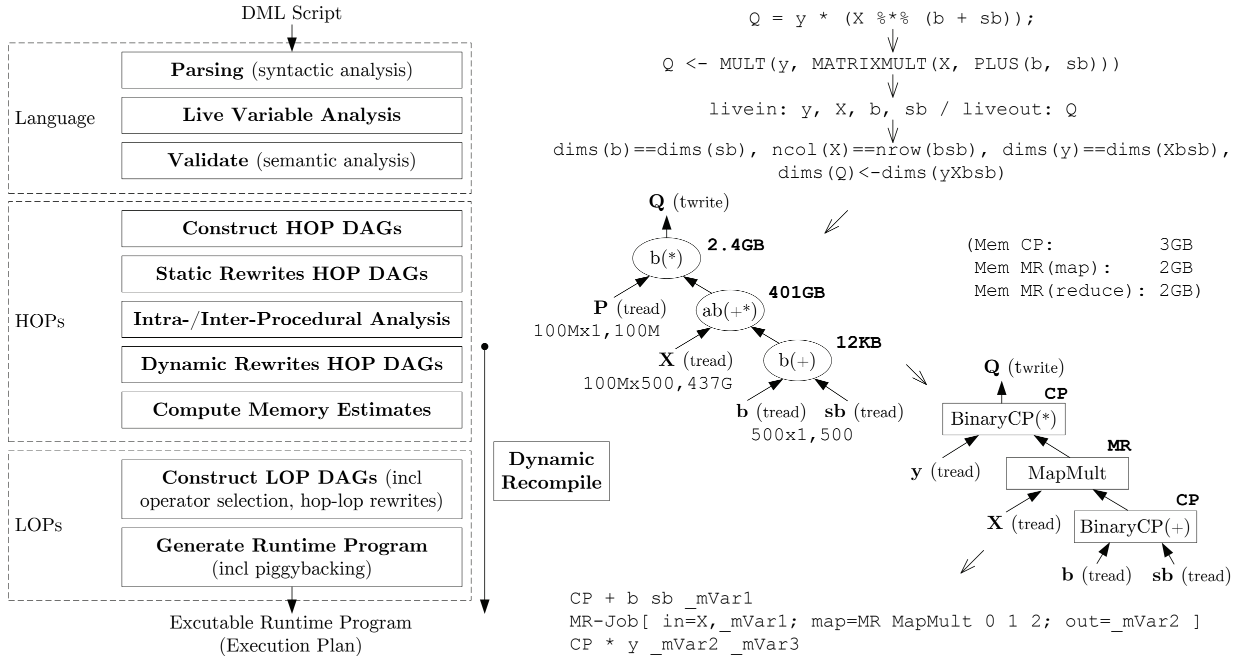

Figure 1: Overview of SystemML's Compilation Chain with Examples.

obtain, for example, livein and liveout variable sets in a forward and backward pass over the entire program. The third compilation step at language level is *validate*, where we perform a semantic analysis of the program and its expressions. In contrast to the previous two steps, this validation is already inherently ML-domain specific. Examples for expression validation are compatibility checks of dimensions and checks for mandatory parameters of built-in functions. This validate step is done in one pass over the entire program, where we recursively validate all statement blocks, statements and expressions. During this pass, we also do an initial constant and size (data characteristic) propagation, with awareness of size updates in conditional branches or loops.

HOP-Level (Subsections 3.1 and 3.2): For each basic block of statements—i.e., predicates and last-level statement blocks—we then create DAGs of high-level operators. Nodes represent (logical) operations and their outputs (scalar/matrix with specific value type), while edges represent data dependencies. First, we recursively *construct* a single operator tree for all statements and expressions of this statement block. Second, after all DAGs have been constructed, we apply *static HOP DAG rewrites*. These rewrites comprise all size-independent rewrites, i.e., program and HOP DAG transformations that are always required or always (assumed to be) beneficial. Examples are mandatory format conversions, common subexpression elimination (CSE), constant folding, algebraic simplifications, and branch removal. Third, we subsequently perform an *intra-/inter-procedural analysis*, where we propagate size information over the—now rewritten—program structure and into functions. Note that we only propagate sizes into functions if they are called with consistent sizes of arguments. Fourth, we apply *dynamic HOP DAG rewrites*, which cover all size-dependent HOP rewrites. These rewrites include (a) simplifications that are always beneficial but only apply under certain size conditions, and (b) cost-based rewrites that require sizes for cost estimation. Examples are matrix multiplication chain optimization and dynamic algebraic simplifications. Fifth, based on propagated sizes, we compute worst-case *memory estimates* for all HOPs. These estimates reflect the memory consumption of this operation with regard to in-memory execution in CP. Figure 1 (middle right) shows the HOP DAG of the example expression annotated with size information and memory estimates. This HOP DAG includes two binary HOPs b(+) and b(*) for cell-wise vector addition and multiplication, respectively, as well as an aggregate binary HOP ab(+*) for matrix multiplication.

LOP-Level (Subsections 3.3 and 3.4): With the rewritten HOP DAGs and computed memory estimates, all preconditions for runtime plan generation are available. We generate runtime plans via DAGs of low-level operators, where nodes represent (physical) operations and edges represent again data dependencies. Individual LOPs are runtime-backend-specific, but a DAG can contain mixed LOP types of integrated runtimes like CP and MR. First, we *construct LOP DAGs* for all HOP DAGs. The compiler uses a recursive traversal (with memoization) to perform this transformation. The recursion, which starts at the DAG root nodes, involves multiple steps per operator: We start with LOP operator selection, where we decide between CP and MR under hard constraints w.r.t. memory estimates and budget as well as on specific physical operators. If sizes are unknown, we fall back to robust MR operators but mark the current HOP for recompilation. Subsequently, we apply specific HOP-LOP rewrites that inherently depend on which physical operator we have chosen. Finally, the individual LOPs are constructed and configured. Second, once the LOP DAG is complete, we *generate the runtime program*. This phase of compilation includes the generation of executable program blocks per statement block and instructions for each LOP DAG. During instruction generation, we also perform piggybacking of multiple MR LOPs into composite MR jobs. Piggybacking is done via a greedy bin packing algorithm that satisfies additional constraints such as map/reduce execution location and memory constraints. Finally, we obtain an executable runtime program, where CP and MR instructions are wrappers of matrix operations which work on entire matrices or matrix blocks respectively. Figure 1 (bottom right) shows the LOP DAG of the example expression. For both binary HOPs, we create `BinaryCP` LOPs as the sizes are small given a CP memory budget of $3\,GB$. For the aggregate binary HOP, however, we create a `MapMult` LOP since the operation memory of $401\,GB$ is too large for the CP memory budget but the vector input ($4\,KB$) fits into the map task budget of $2\,GB$.

Runtime-Level (Section 4): During runtime, we *dynamically recompile* HOP DAGs as required. Recompilation comprises all compilation steps from dynamic rewrites down to runtime instructions as shown in Figure 1 (Dynamic Recompile). This dynamic recompilation is our robust fallback strategy whenever we are not able to propagate size or sparsity during initial compilation, and it is very important due to high latency of unnecessary MR jobs and high impact of selecting the right physical operators on large data. Note that for average scripts, language-level compilation requires about $200\,ms$ per script and HOP-level compilation takes about $10\,ms$ per DAG; while recompilation (including LOP-level) is by design $< 1\,ms$ per DAG because it is part of the critical path. Finally, there are additional runtime optimizers like the `parfor` optimizer [2] that includes task-parallelism-specific rewrites and works with a global optimization scope over the entire loop body.

3 Selected Optimization Phases

We now describe essential optimization phases of SystemML's compilation chain in detail. This description includes (1) static and dynamic HOP DAG rewrites, (2) size propagation and memory estimates, (3) operator selection and HOP-LOP rewrites, as well as (4) runtime plan generation via piggybacking.

3.1 Static and Dynamic Rewrites

Fundamentally, we distinguish two rewrite categories: static (size independent) and dynamic (size dependent), which are both applied at the HOP level to enable reuse across runtime backends. Examples for static rewrites comprise common subexpression elimination, constant folding, static algebraic simplifications, and branch removal. In addition, examples for dynamic rewrites are matrix multiplication chain optimization and dynamic algebraic simplifications. In the following, we discuss these essential rewrites in detail.

Common Subexpression Elimination (CSE): During HOPs construction, we created an initial operator tree. CSE then identifies and removes redundant operations. There are two sources of redundancy: (1) from the original specification (see for example `(1-p)` in Example 2, line 14/15), and (2) resulting from other rewrites.

Our simple yet efficient approach consists of two steps. First, we collect and replace all leaf nodes—i.e. variable reads and literals—via a unique name-operator map. Second, we recursively remove common subexpressions bottom-up starting at these leaf nodes. For each node, we compare and merge parent nodes if they have the same inputs and equivalent configurations. This step is done in an operator-aware fashion in order to prevent invalid eliminations of non-deterministic operators like `rand` with unspecified seed.

Constant Folding: Constant folding mostly serves as a preparation step for size propagation and more sophisticated rewrites. We fold sub-DAGs of operations over literals into a single literal by—similar to existing work [1]—compiling and executing runtime instructions. This elegant approach ensures consistency, which is especially important with regard to type promotion as well as NaN and overflow handling.

Static Algebraic Simplifications: Static algebraic simplifications are size-independent rewrites, since they are assumed to be always beneficial. As shown in Equation 5, this category of rewrites includes the removal of unnecessary operations, the transformation from binary to unary operations because they are easier to parallelize, and specific simplifications that either reduce the number of intermediates or change the asymptotic complexity:

$$\begin{aligned}
\text{Remove Unnecessary Operations:}\quad & \mathbf{X}^{\top\top},\quad \mathbf{X}/1,\quad \mathbf{X}\cdot 1,\quad \mathbf{X}-0 \rightarrow \mathbf{X},\quad matrix(1,\ldots))/\mathbf{X} \rightarrow 1/\mathbf{X} \\
& rand(\ldots,\text{min}=-1,\text{max}=1)\cdot 7 \rightarrow rand(\ldots,\text{min}=-7,\text{max}=7) \\
\text{Binary to Unary:}\quad & \mathbf{X}+\mathbf{X} \rightarrow 2\cdot\mathbf{X},\quad \mathbf{X}\cdot\mathbf{X} \rightarrow \mathbf{X}^2,\quad \mathbf{X}-\mathbf{X}\cdot\mathbf{Y} \rightarrow \mathbf{X}\cdot(1-\mathbf{Y}) \\
\text{Simplify Diag Aggregates:}\quad & sum(diag(\mathbf{X})) \rightarrow trace(\mathbf{X}),\quad trace(\mathbf{XY}) \rightarrow sum(\mathbf{X}\cdot\mathbf{Y}^{\top})
\end{aligned} \tag{5}$$

We apply those rewrites after CSE because—especially for binary to unary rewrites—this order greatly simplifies the rewrite framework since we only need to probe local inputs for identity. Finally, we do a second CSE pass.

Remove Branches: As a preparation for size propagation, we remove branches with constant conditions. This rewrite happens after constant folding, which folds arbitrary constant predicate expressions into a boolean constant. In case of true conditions, we replace the entire if-else block with the if branch body, while for false conditions, we either use the else branch body or remove again the entire block. Removing, for instance, branches that append a column of 1s to $\mathbf{X}$ for models with intercept allows us to propagate unconditional sizes.

Matrix Multiplication Chain Optimization: The associative order of a chain of matrix multiplications strongly affects sizes of intermediates and computational effort. Consider for example $\mathbf{X}^{\top}(diag(\mathbf{v})\mathbf{Xd})$ from Example 2, line 5: computing $\mathbf{X}^{\top}diag(\mathbf{v})$ first would produce an intermediate in the size of $\mathbf{X}$, while performing these operations in the right order produces only vector intermediates. Since this optimization problem exhibits the properties of *optimal substructure* and *overlapping subproblems*, we use a classic dynamic programming algorithm. This bottom-up algorithm computes the cost-optimal parentheses for subchains and composes them to the overall chain, where we currently assume independence with regard to the sparsity of intermediates.

Dynamic Algebraic Simplification Rewrites: Dynamic algebraic simplifications are size dependent rewrites, i.e., rewrites which use sizes in validity constraints or for cost comparison. This category includes the removal of unnecessary indexing operations, simplifications for structural aggregations, the removal of operations with empty output, and simplifications of operations involving vectors.

$$\begin{aligned}
\text{Remove Unnecessary Indexing:}\quad & \mathbf{X}=\mathbf{Y}[\ldots],\quad \mathbf{X}[\ldots]=\mathbf{Y} \rightarrow \mathbf{X}=\mathbf{Y}, \text{ iff } \text{dims}(\mathbf{X})=\text{dims}(\mathbf{Y}) \\
& \mathbf{X}=\mathbf{Y}[\ldots] \rightarrow \mathbf{X}=matrix(0,\ldots), \text{ iff } \text{nnz}(\mathbf{Y})=0 \\
\text{Simplify Aggregates:}\quad & colSums(\mathbf{X}),\quad rowSums(\mathbf{X}) \rightarrow sum(\mathbf{X}), \text{ iff } \text{ncol/nrow}(\mathbf{X})=1 \\
\text{(sum, min, max, mean, prod)}\quad & colSums(\mathbf{X}),\quad rowSums(\mathbf{X}) \rightarrow \mathbf{X}, \text{ iff } \text{nrow/ncol}(\mathbf{X})=1 \\
& sum(\mathbf{X}) \rightarrow 0,\quad colSums(\mathbf{X}) \rightarrow matrix(0,\ldots), \text{ iff } \text{nnz}(\mathbf{X})=0 \\
\text{Remove Empty Operators:}\quad & \mathbf{X}+\mathbf{Y},\quad \mathbf{X}-\mathbf{Y} \rightarrow \mathbf{X}, \text{ iff } \text{nnz}(\mathbf{Y})=0 \\
& \mathbf{XY},\quad \mathbf{X}\cdot\mathbf{Y},\quad round(\mathbf{X}),\quad \mathbf{X}^{\top} \rightarrow matrix(0,\ldots), \text{ iff } \text{nnz}(\mathbf{X}|\mathbf{Y})=0 \\
\text{Simplify Matrix Multiply:}\quad & diag(\mathbf{X})\mathbf{y} \rightarrow \mathbf{X}\cdot\mathbf{y}, \text{ iff } \text{ncol}(\mathbf{y})=1, \text{ otherwise } (\mathbf{X}matrix(1,...))\cdot\mathbf{y} \\
& diag(\mathbf{XY}) \rightarrow rowSums(\mathbf{X}\cdot t(\mathbf{Y})), \text{ iff } \text{ncol}(\mathbf{y})=1 \\
& (-\mathbf{X}^{\top})\mathbf{y} \rightarrow -(\mathbf{X}^{\top}\mathbf{y}), \text{ iff } \text{size}(\mathbf{X}^{\top}\mathbf{y}) < \text{size}(\mathbf{X})
\end{aligned} \tag{6}$$

If sizes are known during initial compilation, those rewrites are applied statically (in-place of the original DAG). Otherwise those rewrites take affect during dynamic recompilation with potential for a very aggressive rewrite scope. For instance, in Example 2, each outer iteration re-initializes sb to an empty matrix. Hence, in each first inner iteration, we can eliminate `2*sum(sb*d)` and `sum(sb^2)` (line 7) via dynamic rewrites.

3.2 Size Propagation and Memory Estimates

Size propagation and memory estimates are crucial for declarative ML with both in-memory and large-scale computation. Any memory estimates inherently need to reflect the underlying runtime. In our single node runtime, a multi-level buffer pool controls in-memory objects, and runtime instructions pin their inputs/outputs in memory. This pinning prevents serialization in operations like matrix multiply that access data multiple times or out-of-order. However, to prevent out-of-memory situations, we need to guarantee that memory estimates never underestimate. We now describe size propagation over ML programs and worst-case memory estimates.

Intra-/Inter-Procedural Analysis (IPA): Important size information for memory estimates are matrix dimensions and sparsity. Input sizes of `read`s are given by meta data that is mandatory for sparse matrix formats. Both *validate* and *IPA* then propagate these sizes over the entire ML program. Here, we use IPA as an example. IPA aims at propagating sizes into and across DML-bodied functions. In general, we use a candidate-based algorithm. First, we determine candidate functions by collecting all function calls and their input parameters. Second, we prune all functions that are called with potentially different dimension sizes of input matrices. Third, for all remaining positive candidates, we determine if we can also safely propagate sparsity into that function. Fourth, we do a full intra-/inter-procedural analysis over the entire program. In detail, we iterate over the hierarchy of statement blocks. For each statement block, we push input variable sizes into DAG leaf nodes, recursively propagate them bottom-up and in an operator-aware manner through the DAG, and finally extract the result variables of this DAG. IPA also pays special care to conditional control structures. For if conditions, we only propagate the size of a variable if both branches lead to the same size. For loops, we determine if sizes change in the loop body; and if they do, we re-propagate unknown sizes into the loop body. Whenever, we hit a function call that is marked as an IPA candidate, we recursively propagate sizes into that function and after that continue intra-procedure analysis with the resulting function output sizes.

Memory Estimates: We then recursively compute memory estimates—-with memoization—for entire HOP DAGs. These estimates reflect single-threaded, in-memory computations. Starting at leaf nodes (reads and literals), we compute the output memory estimate according to the HOP output sizes via a precise model of our sparse and dense matrices [2]. If the sparsity is unknown, we use dense as the worst case since we only switch to sparse if smaller. We then compute output estimates for all non-leaf HOPs, after which we compute operation memory estimates using the sum of all child outputs, intermediates, and output estimate. For example, the operation memory estimate of a binary operator for cell-wise matrix addition would include the output memory of both inputs and its own output. During this process, we also propagate worst-case sparsity and dimensions according to the operator semantics. For example, for a $[m \times k, s_1] \times [k \times n, s_2]$ matrix multiply, we use a worst case sparsity estimate of $s_3 = \min(1, s_1 k) \cdot \min(1, s_2 k)$ although the average case estimate would be $s_3 = 1 - (1 - s_1 s_2)^k$. Additionally, we backtrack size constraints. For example, for row-wise left indexing into a matrix R, the input matrix size is known to be of size $[1 \times ncol(R)]$. Finally, these memory estimates are used for all in-memory matrices, matrix blocks, and operations over all of them.

3.3 Operator Selection and Ordering

Operator selection determines the best execution strategy for every HOP in a DAG. The selected plan is a sub-DAG of LOPs with certain physical properties but we retain the HOP DAG for later recompilation. Operator selection includes (1) selecting the execution type, (2) selecting execution-type-specific physical operators, and

(3) local rewrites that are dependent on the physical operator or computed memory estimates.

LOP Selection (Execution Type and Physical Operators): The translation from HOPs to LOPs is done in a local manner on a per-HOP basis. For a given HOP, the selection of LOPs is determined by four factors: (1) memory budget for CP, (2) memory estimate of the HOP, (3) data characteristics such as dimensions and sparsity, and (4) cluster characteristics such as degree of parallelism and memory budget of mappers/reducers. Based on the HOP memory estimate, we first decide on the execution type, i.e. single-node CP or MR. Currently, we follow the heuristic that in-memory operations require less time than their MR counterparts and hence, select CP whenever the memory estimate is smaller than the budget. This heuristic can be relaxed by incorporating cost models of execution time for generated runtime plans. Second, we determine the exact runtime strategy. For example, we currently support five physical operators for MR matrix multiplication: `TSMM` (for $\mathbf{X}^\top\mathbf{X}$), `MapMult` (for $\mathbf{Xv}$, if $\mathbf{v}$ fits into the mapper memory budget), `MapMultChain` (for entire chains of $\mathbf{X}^\top(\mathbf{w} \cdot \mathbf{Xv})$ or $\mathbf{X}^\top(\mathbf{Xv})$ if $\mathbf{w}$ and $\mathbf{v}$ fit into the mapper memory budget), as well as `CPMM` and `RMM` (as previously described in [5]). The decision on physical operators is governed by a combination of cost functions and heuristics, where it is important to take the resulting degree of parallelism into account. For common use cases like Examples 1 and 2, we typically compile `TSMM`, `MapMult`, or `MapMultChain`, which nicely fit the MR model in terms of core computation in mappers, local aggregation via combiners, and very small final aggregation.

HOP-LOP Rewrites: After the decision on physical operators, we apply HOP-LOP rewrites that depend on the chosen LOPs. Examples are decisions on MR aggregation, empty block materialization, and input partitioning but also rewrites like transpose re-ordering. Let us use $\mathbf{X}^\top\mathbf{y}$ from Example 1 (line 2), where $\mathbf{X}$ is $[10^8 \times 500]$ and assume we decided for MR MapMult. First, since multiple row blocks (10^5 for a blocksize of $[10^3 \times 10^3]$) contribute to the result, we need to add another LOP for local aggregation. Second, since $\mathbf{y}$ requires already $800\,MB$ and needs to be read sideways through distributed cache, we introduce a LOP for data partitioning. If $\mathbf{y}$ fits in the CP memory budget this is done in CP; otherwise in MR. Third, we reorder $\mathbf{X}^\top\mathbf{y}$ to $(\mathbf{y}^\top\mathbf{X})^\top$ if $\mathbf{y}$ and the result are smaller than $\mathbf{X}$ and both introduced transpose operations fit in the CP memory budget. The partition LOP is accordingly configured as column block partitioning. Fourth, because the output of MapMult is consumed only by the CP transpose, we configure the MapMult to not output empty blocks. In summary, even for a single HOP, there are many different execution plans, depending on data and cluster characteristics.

3.4 Piggybacking

We generate the runtime program via piggybacking. The input to this compilation step is the previously constructed LOP DAG and the output is an ordered list of runtime instructions. This list of instructions includes in-memory CP instructions and MR-Job instructions. Piggybacking packs multiple MR instructions into shared MR jobs for scan sharing (data-dependent) and latency reduction (independent). Currently, our piggybacking optimization objective is to minimize the number of MR jobs, but other objectives like the amount of read/written data are possible. In the following, we describe the core algorithm (an extension of the previous description [5]).

Piggybacking MR LOPs into MR Jobs: Conceptually, the problem of packing instructions into a minimal number of MR jobs is a bin packing problem with multiple constraints. Example constraints are (1) job-type requirements, (2) the execution location of operations (map, reduce, map and/or reduce, CP), (3) if the operation changes keys (breaks block alignment), (4) if the operation groups keys (aligns blocks), and (5) the memory consumption. Our algorithm has two steps. First, we do a topological sort of the LOP DAG. Second, we do a greedy grouping of operations via an extended *next fit* bin packing heuristic. The algorithm operates in multiple *rounds*. In each round, it maintains multiple open MR jobs, one for each job type. It subsequently assigns LOPs to these open MR jobs as long as all constraints are satisfied. Once all possible LOPs are assigned in the current round, the MR jobs are closed and corresponding MR-Job instructions are generated. We repeat this process until all LOPs are assigned. This algorithm has linear best-case and quadratic worst-case complexity in the number of LOPs and thus allows for efficient runtime plan generation.

4 Dynamic Recompilation

The described compilation techniques work very well if we can infer intermediate sizes and sparsity, which is true for simple iterative algorithms where the training data $\mathbf{X}$ is accessed read-only and thus all important operations are known. For other types of programs, the sizes or sparsity of intermediates may be unknown during initial compilation. We use dynamic recompilation as our robust fallback strategy for these cases.

Example scenarios are scripts with functions, sampling, data dependent operations, and changing dimensions or sparsity. First, if functions are called with arguments of different sizes, the plan of the function itself, the function outputs, and subsequent operations are unknown. The same limitation also applies to external UDFs. Second, sampling—as used for fold creation in cross validation or pre-sampling large data—leads to unknowns for the entire training algorithm as well. For instance, consider a simple sampling via permutation matrices:

```
1:   s  = ppred( rand(rows=nrow(X), cols=1, min=0, max=1), 0.01, "<=" );
2:   Xs = removeEmpty( diag(s), margin="rows" ) %*% X;  #take an ~1% sample of X
```

where `ppred` creates a sample indicator vector $\mathbf{s}$. This allows us to sample $\mathbf{X}$ with a single matrix multiplication but due to randomization leads to an unknown number of rows of the output matrix $\mathbf{Xs}$. Third, data-dependent operations like `table` (computes contingency tables), `aggregate` (computes grouped aggregates) or `removeEmpty` (removes empty rows or columns) pose challenges for size inference because even the dimensions become unknown or are heavily overestimated. Fourth, changing dimensions or sparsity also lead to unknowns because we need to conservatively consider the worst case.

Dynamic recompilation inherently shares the same goals and challenges as adaptive query processing [3] in DBMSs. Specifically, dynamic recompilation is always a trade-off between the potential overhead of breaking pipelines and benefit of re-optimization. However, we can exploit specific characteristics of large-scale ML programs. First, conditional control flow requires us to split DAGs anyway into statement blocks with materialized intermediates. Second, the re-optimization potential is huge. Operations on small data execute in milliseconds in memory while we would pay 10s latency for MR jobs. In addition, picking the right physical operators is crucial on large data as well. These characteristics allow a robust and simple, yet very effective recompiler that is used by default. In what follows, we describe (1) compile time decisions, and (2) recompilation during runtime.

4.1 Optimizer Recompilation Decisions

Dynamic recompilation works at the granularity of HOP DAGs in order to exploit natural block boundaries and simplify compilation to unconditional blocks. The optimizer makes two important decisions, namely (1) on the recompilation granularity by splitting HOP DAGs, and (2) on whether or not to mark HOP DAGs for recompilation. This control allows us to trade the flexibility of an interpreter with the efficiency of a compiler.

Split HOP DAGs for Recompilation: Our major goal is to prevent unknowns but keep DAGs as large as possible to exploit all piggybacking opportunities. Hence, we follow a very conservative approach of splitting HOP DAGs for recompilation: by default, we do not split any DAGs. Exceptions are persistent reads with unknown sizes (without provided metadata) and specific data-dependent operations like `table` where our worst-case estimates largely overestimate. This DAG split is done either during initial parsing or as a static HOP rewrite rule if dependent on additional information. In detail, we collect these operators and their inputs, replace them with transient reads and put them into a new statement block which is inserted just before the original one.

Mark HOP DAGs for Recompilation: Marking DAGs for recompilation is part of operator selection during LOP DAG construction. Our simple strategy is to mark a HOP for recompilation whenever we select conservative physical operators (MR, or CP out-of-core) due to unknown sizes or sparsity. This means that most DAGs that include at least one MR job are marked for recompilation. This policy is advantageous in any case because the large latency of an MR job or disk I/O makes recompilation overhead negligible. For large data this policy allows the compiler to change physical operators, and for small data we might be able to compile pure in-memory

instructions. We do not mark CP operations for recompilation—even if the sparsity is unknown—because it is unknown if the overhead of recompilation can be amortized by the benefits of dynamic rewrites. Subsequently, we aggregate recompilation flags of all HOPs to the DAG level: if at least one HOP requires recompilation, the entire DAG requires recompilation and we materialize this information.

Finally, note that the recompiler is used for consistency in many related optimization phases. For example, we currently exploit the recompiler for size propagation in intra-/inter procedure analysis, as a preparation step in runtime optimizers like `parfor`, and for cost estimation of runtime plans in more sophisticated optimizers.

4.2 Dynamic Recompilation at Runtime

Dynamic recompilation during runtime is done at specific recompilation hooks. The most important hooks exist before execution of last-level program block instructions and predicate instructions. Just before executing the initially compiled instructions, we access the statement block associated with a program block; if it is marked for recompilation, we compile the HOP DAG by leveraging the current symbol table with metadata and statistics of all live variables. In detail, dynamic recompilation of a single HOP DAG comprises the following steps:

- **Deep Copy DAG:** First, we create a deep copy of the given HOP DAG in order to apply non-reversible dynamic rewrites but keep the original DAG for later recompilations.
- **Update DAG Statistics:** Second, we update DAG statistics by updating leaf node statistics according to the current symbol table. Subsequently, we recursively propagate those statistics bottom-up through the DAG, where each HOP updates its output statistics according to its input statistics and semantics. For operators with size expressions (e.g., `rand`, `seq`, `reshape`, range indexing) we also evaluate simple expressions similar to constant folding but with regard to the current values of the symbol table.
- **Dynamic Rewrites:** Third, we apply the dynamic rewrites described earlier. Having the symbol table with exact statistics allows us to apply very aggressive rewrites that would be invalid in general. Examples are first iterations of certain algorithms with zero initialized model coefficients.
- **Recompute Memory Estimates:** Fourth, we recompute the memory estimates according to the updated statistics and rewritten DAG. With this unconditional scope of a single DAG, we typically obtain very good estimates because we already split DAGs where we would heavily overestimate.
- **Generate Runtime Instructions:** Fifth, we construct LOPs (including operator selection and HOP-LOP rewrites) and generate runtime instructions (including piggybacking). All these substeps are heavily dependent on the memory estimates and hence benefit from updated statistics.

5 Conclusions

We showed an overview of SystemML's compilation chain, selected optimization phases, and dynamic recompilation. Declarative large-scale ML with a high-level domain-specific language aims at both (1) flexibility in specifying new large-scale machine learning algorithms as well as (2) efficiency and scalability via size-aware automatic optimization. Inherently, the optimizer is key because it allows users to write their algorithms once and deploy them in many settings with different or even unknown input characteristics. Our major directions for future work comprise (1) optimizer support for next generation runtime platforms like YARN and Spark, (2) optimizer support for hardware accelerators like many core co-processors or GPUs, and (3) global data flow optimization. We strongly believe that both growing analysis complexity and growing data sizes will further increase the importance of automatic optimization. Hence, we would like to encourage the database community to contribute to this emerging domain of declarative machine learning.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, & Tools*. Addison-Wesley, 2007.

[2] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[3] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[4] Dmitriy Lyubimov. *Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines*. The Apache Software Foundation, 2014. `http://mahout.apache.org/users/ sparkbindings/home.html`.

[5] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[6] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[7] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[8] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust Region Newton Method for Logistic Regression. *Journal of Machine Learning Research*, 9:627–650, 2008.

[9] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for Distributed Machine Learning. In *ICDM*, 2013.

[10] The Apache Software Foundation. *Mahout*.

[11] Y. Tian, S. Tatikonda, and B. Reinwald. Scalable and Numerically Stable Descriptive Statistics in SystemML. In *ICDE*, 2012.