# SystemML: Declarative Machine Learning on Spark

Matthias Boehm[1], Michael W. Dusenberry[2], Deron Eriksson[2], Alexandre V. Evfimievski[1], Faraz Makari Manshadi[1], Niketan Pansare[1], Berthold Reinwald[1], Frederick R. Reiss[1,2], Prithviraj Sen[1], Arvind C. Surve[2], Shirish Tatikonda[1*]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] IBM Spark Technology Center; San Francisco, CA, USA

## ABSTRACT

The rising need for custom machine learning (ML) algorithms and the growing data sizes that require the exploitation of distributed, data-parallel frameworks such as MapReduce or Spark, pose significant productivity challenges to data scientists. Apache SystemML addresses these challenges through declarative ML by (1) increasing the productivity of data scientists as they are able to express custom algorithms in a familiar domain-specific language covering linear algebra primitives and statistical functions, and (2) transparently running these ML algorithms on distributed, data-parallel frameworks by applying cost-based compilation techniques to generate efficient, low-level execution plans with in-memory single-node and large-scale distributed operations. This paper describes SystemML on Apache Spark, end to end, including insights into various optimizer and runtime techniques as well as performance characteristics. We also share lessons learned from porting SystemML to Spark and declarative ML in general. Finally, SystemML is open-source, which allows the database community to leverage it as a testbed for further research.

## 1. INTRODUCTION

Data scientists are challenged in today's data economy due to (1) the need for very sophisticated, custom machine learning (ML) algorithms—beyond off-the-shelf library algorithms, (2) the growing amount of data, partially spurred by the Internet of Things (IoT) in industries such as manufacturing, mobile, automotive, and health care, and (3) the resulting need to run these custom ML algorithms on distributed, data-parallel systems such as MapReduce (MR) [14], Spark [41], or Flink [2] for scalability and performance on cost-efficient commodity clusters.

**Overview of SystemML:** Apache SystemML [6] addresses these challenges through declarative ML [10] that aims at a high-level specification of ML algorithms to simplify the development and deployment of ML algorithms by separating algorithm semantics from underlying data representations and runtime execution plans. This separation provides tremendous benefits and opportunities, such as (1) a data-scientist-centric algorithm specification which improves the productivity of data scientists, (2) algorithm reusability and simplified deployment for varying data characteristics and runtime environments, and (3) automatic optimization of runtime execution plans. Data scientists express their custom ML algorithms in a domain-specific language with precise semantics as well as abstract data types and operations, independent of their implementation. SystemML's language is expressive enough to cover a broad class of ML algorithms: descriptive statistics, classification, clustering, regression, matrix factorizations, dimensions reduction, and survival models for training and scoring. Generally, algorithms that can be expressed using vectorized operations are a good fit for SystemML. The SystemML cost-based compiler automatically generates hybrid runtime execution plans that are composed of single-node and distributed operations depending on data and cluster characteristics such as data size, data sparsity, cluster size, memory configurations, while exploiting the capabilities of underlying data-parallel frameworks such as MR or Spark.

**SystemML on Spark:** SystemML started off with an implementation on MR as the data-parallel framework dejour [15] in order to share cluster resources with other MR-based systems, and later evolved to utilize the more general YARN [36]. Apache Spark [5] further offered a number of advantages over MR such as a unification of SQL, graph, stream, and ML processing by providing a common RDD data structure, a general DAG execution engine with lazy evaluation, and distributed in-memory caching. These capabilities make Spark particularly attractive for ML, which often requires custom data preparation and repeated, read-only data access in iterative ML algorithms. However, implementing ML algorithms directly against Spark compromises the benefits of declarative ML and requires substantial effort. Hence, we decided to fit SystemML into the Spark ecosystem by (1) providing Spark APIs for a seamless integration, and (2) automatically compiling ML algorithms into efficient execution plans on top of Spark. Declarative ML allowed us to transition from MR to YARN and Spark as well as to support these frameworks simultaneously without the need for any ML algorithm changes.

**Contributions:** Our major contribution is an end-to-end description of declarative ML on Spark. We share lessons learned, describe implementation choices, introduce several techniques that tackle challenges of memory handling and lazy evaluation, and discuss performance insights through

[*] Now at Target Corp.; Work done at IBM Research – Almaden.

*Proceedings of the VLDB Endowment,* Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

an empirical evaluation of end-to-end ML algorithms. Our detailed contributions reflect the structure of this paper:

- *Background:* We give an up-to-date overview of SystemML as a representative system for declarative ML in Section 2, and introduce our running example in the context of the SystemML's Spark APIs.
- *Optimizer Integration:* We explain optimizer extensions for our Spark backend in Section 3. This covers rewrites, memory estimates and constraints, operator selection, as well as parfor optimizer extensions.
- *Runtime Integration:* We describe the Spark runtime backend in Section 4. This includes matrix representations, the buffer pool integration, partitioning-preserving operations, as well as specific optimizations.
- *Experiments:* We present end-to-end experimental results for a variety of ML algorithms, data characteristics, and spark-specific optimizations in Section 5.
- *Discussion and Related Work:* Finally, we discuss lessons learned from rebasing SystemML on Spark in Section 6, and compare SystemML to related work in the larger context of large-scale ML in Section 7.

# 2. BACKGROUND

In this section, we provide a current overview of SystemML and its high-level architecture with various execution backends, including Spark. We introduce our running example in the context of the Spark ecosystem. This example is used throughout the paper to explain declarative ML, APIs, optimizations, runtime, and performance insights.

## 2.1 SystemML Architecture

ML algorithms are expressed in a declarative, high-level language, called DML (Declarative ML). SystemML compiles and optimizes these algorithms into hybrid runtime plans of multi-threaded, in-memory operations on a single node (scale-up) and distributed MR or Spark operations on a cluster of nodes (scale-out). SystemML's high-level architecture consists of the following components.

**Language:** DML (with either R- or Python-like syntax) provides linear algebra primitives, a rich set of statistical functions and matrix manipulations, as well as user-defined and external functions, control structures including parfor loops, and recursion [6]. The language component parses a given DML script into a hierarchy of statement blocks and statements as defined by control structures, and performs syntactic analysis, live variable analysis, and semantic validation (e.g., matching matrix dimensions). During that process we also retrieve input data characteristics—i.e., format, number of rows, columns, and non-zero values—as well as infrastructure characteristics, which are used for subsequent optimizations. Finally, we construct directed acyclic graphs (DAGs) of high-level operators (HOPs) per statement block.

**Optimizer:** The SystemML optimizer [8, 9, 19] works over programs of HOP DAGs, where HOPs are operators on matrices or scalars, and are categorized according to their access patterns. Examples are matrix multiplications, unary aggregates like `rowSums()`, binary operations like cell-wise matrix additions, reorganization operations like transpose or sort, and more specific operations. We perform various optimizations on these HOP DAGs, including algebraic simplification rewrites, intra-/inter-procedural analysis for statistics propagation into functions and over entire programs, and operator ordering of matrix multiplication chains. We compute memory estimates for all HOPs, reflecting the memory requirements of in-memory single-node operations and intermediates. Each HOP DAG is compiled to a DAG of low-level operators (LOPs) such as grouping and aggregate, which are backend-specific physical operators. Operator selection picks the best physical operators for a given HOP based on memory estimates, data, and cluster characteristics. Individual LOPs have corresponding runtime implementations, called instructions, and the optimizer generates an executable runtime program of instructions.

**Runtime:** We execute the generated runtime program locally in CP (control program), i.e., within a driver process. This driver handles recompilation, runs in-memory single-node CP instructions (some of which are multi-threaded), maintains an in-memory buffer pool, and launches MR or Spark jobs if the runtime plan contains distributed computations in the form of MR or Spark instructions [8, 15]. For the MR backend, the SystemML compiler groups LOPs—and thus, MR instructions—into a minimal number of MR jobs (MR-job instructions), whereas for the Spark backend, we rely on Spark's lazy evaluation and stage construction. CP instructions may also be backed by GPU kernels [7]. The multi-level buffer pool caches local matrices in-memory, evicts them if necessary, and handles data exchange between local and distributed runtime backends. Both CP and distributed operations for descriptive statistics and aggregations are numerically stable [35]. The core of SystemML's runtime instructions is an adaptive matrix block library, which is sparsity-aware and operates on the entire matrix in CP, or blocks of a matrix in a distributed setting. Further key features include parallel for-loops for task-parallel computations [8], and dynamic recompilation for runtime plan adaptation addressing initial unknowns [9].

## 2.2 Running Example

As our running example, we introduce a DML script of the popular alternating least squares (ALS) algorithm for matrix completion [43]. In the context of collaborative filtering in recommender systems, ALS tries to decompose a partially observed user-item-rating matrix $\mathbf{X}$ ($m \times n$) into two factor matrices $\mathbf{U}$ ($m \times r$ representing user factors) and $\mathbf{V}$ ($r \times n$ representing item factors) of low rank $r \ll \min(m, n)$, such that $\mathbf{X} \approx \mathbf{UV}$. In our example, the quality of the approximation is measured as L2 regularized squared loss over the observed entries of $\mathbf{X}$ with $L = \sum_{i,j} \mathbf{W}_{ij}(\mathbf{X}_{ij} - [\mathbf{UV}]_{ij})^2 + \lambda \left(\|\mathbf{U}\|_{\mathrm{F}}^2 + \|\mathbf{V}\|_{\mathrm{F}}^2\right)$, where $\|\cdot\|_{\mathrm{F}}$ is the Frobenius norm and $\mathbf{W}_{ij} = |\mathrm{sgn}\, \mathbf{X}_{ij}|$. ALS repeatedly keeps one of the unknown matrices fixed and optimizes the other; when fixing $\mathbf{U}$ or $\mathbf{V}$, we obtain a quadratic least-squares problem, one per row (column) of $\mathbf{U}$ ($\mathbf{V}$) with a globally optimal solution. To solve these least-squares problems jointly, we use the conjugate gradient method.

**Example ALS-CG Script:** Below DML script implements the outlined ALS algorithm. It reads an input matrix `X` (line 1). After initializing the parameters (lines 2-6), we compute the user and item factors in two nested loops: (1) the outer while loop repeats alternating minimization of factor matrices (`U` if `is_U` is true, `V` otherwise) for a fixed number of iterations `mi`; (2) the inner while loop performs conjugate gradient steps to optimize one of the factor matrices at a time. In R syntax, we write `%*%` for matrix multiplication and * for element-wise multiplications. Finally, we output the fitted factor matrices U and V (lines 32-33).

```
 1: X = read($inFile);
 2: r = $rank; lambda = $lambda;
 3: U = rand(rows=nrow(X), cols=r, min=-1.0, max=1.0);
 4: V = rand(rows=r, cols=ncol(X), min=-1.0, max=1.0);
 5: W = (X != 0);
 6: mi = $maxiter; mii = r; i = 0; is_U = TRUE;
 7: while(i < mi) {
 8:   i = i + 1; ii = 1;
 9:   if (is_U)
10:      G = (W * (U %*% V - X)) %*% t(V) + lambda * U;
11:   else
12:      G = t(U) %*% (W * (U %*% V - X)) + lambda * V;
13:   norm_G2 = sum(G ^ 2); norm_R2 = norm_G2;
14:   R = -G; S = R;
15:   while(norm_R2 > 10E-9 * norm_G2 & ii <= mii) {
16:     if (is_U) {
17:       HS = (W * (S %*% V)) %*% t(V) + lambda * S;
18:       alpha = norm_R2 / sum (S * HS);
19:       U = U + alpha * S;
20:     } else {
21:       HS = t(U) %*% (W * (U %*% S)) + lambda * S;
22:       alpha = norm_R2 / sum (S * HS);
23:       V = V + alpha * S;
24:     }
25:     R = R - alpha * HS;
26:     old_norm_R2 = norm_R2; norm_R2 = sum(R ^ 2);
27:     S = R + (norm_R2 / old_norm_R2) * S;
28:     ii = ii + 1;
29:   }
30:   is_U = ! is_U;
31: }
32: write(U, $outUFile, format = "text");
33: write(V, $outVFile, format = "text");
```

**Script Customization:** The above DML script uses matrix W as an indicator for observed entries in X (line 5). In other application scenarios, W may represent weights for the entries in X, and may be dense. For example, let W be an outer product of two weight vectors w1 and w2, i.e., W = w1 %*% t(w2). Lines 10, 12, 17, and 21 in the above script would not adequately exploit the modified W, because we compute U %*% V for every cell where $W_{ij} \neq 0$. This can be circumvented by applying the following matrix equalities:

```
# Replace U with S for Line 17
(W * (U %*% V)) %*% t(V) = (U*w1) %*% (V*t(w2)) %*% t(V)
# Replace V with S for Line 21
t(U) %*% (W * (U %*% V)) = t(U) %*% (U*w1) %*% (V*t(w2))
```

where, for example, U*w1 denotes a matrix-vector element-wise multiplication that logically replicates w1 for every column in U. SystemML is now able to exploit the matrix-product associativity, and order the multiplications in a way that avoids dense intermediates in the size of X. A data scientist may simply apply the outlined changes to customize the above ALS algorithm to the needs of her application scenario, whereas changing a tuned large-scale ALS algorithm implementation would be a non-trivial endeavor.

**MLContext API:** SystemML provides an API called MLContext (in Scala, Java, and Python), that allows the user to register RDDs and DataFrames (previously created through Spark SQL or other libraries) as input and output variables of a DML script. This enables SystemML to seamlessly integrate into the entire Spark ecosystem. The following Python example code uses PySpark to read the MovieLens dataset to produce and manipulate a Spark DataFrame (lines 2). The MLContext API allows to register and pass the DataFrame to the ALS algorithm (line 4), invoke the ALS algorithm with parameters (line 7), get the output factor matrices as DataFrames (lines 8), and invoke a DML script for scoring (line 9, script not shown). We use Spark SQL to perform further data post-processing (lines 12/13).

```
 1: from SystemML import MLContext
 2: X = csvReader.load("ratings.csv").drop("timestamp")
 3: ml = MLContext(sc)
 4: ml.registerInput("X", X)
 5: ml.registerOutput("U", U)
 6: ml.registerOutput("V", V)
 7: outputs = ml.execute("ALS.dml", params)
 8: U = outputs.getDF(sqlContext, "U") ...
 9: outputs1 = ml.execute("ALS_predict.dml", params)
10: predictIds = outputs1.getDF("OutP")
11: movies = csvReader.load("movies.csv")
12: prediction = movies.join(
13:    predictIds.C1==movies.movieId).select("title")
```

SystemML provides further APIs, all of which can use exactly the same DML script, which simplifies deployment.

# 3. OPTIMIZER INTEGRATION

Given the DML scripts passed in through the various APIs, SystemML automatically generates efficient execution plans [9]. Optimizer extensions at different levels are required to exploit Spark in a robust and effective way. In this section, we describe Spark-specific rewrites, the handling of memory budgets and constraints, physical operator selection, as well as parfor optimizer extensions.

## 3.1 Spark-Specific Rewrites

Data independence is one of the major goals of declarative ML [10]. Hence, in contrast to other DSLs [24], SystemML does not expose physical properties like caching and partitioning and thus, needs to handle these automatically. We introduce Spark-specific rewrites which address decisions on distributed caching, checkpointing, and partitioning.

**Caching/Checkpoint Injection:** Spark allows an RDD to be persisted into various storage levels for distributed caching. We introduce two simple rewrite rules for injecting these checkpoints, similar but less aggressive than in other systems [3]. By default, we use a storage level MEMORY_AND_DISK in order to exploit caching without deserialization, and to prevent repeated lazy evaluation of expensive operations. First, we inject checkpoints after every persistent read or reblock (e.g., conversion of text to binary block) in order to prevent repeated read from HDFS, text parsing and shuffle. Second, we inject checkpoints before loops for all read-only variables in the loop body. Only for parfor loop bodies, checkpoint injection is deferred until parfor optimization. These checkpoints also bring matrices into read optimized form. We coalesce the number of partitions according to the HDFS block size if necessary. Further, for sparse matrices, we convert matrix blocks into a memory-efficient CSR representation, and for ultra-sparse matrices, we use the storage level MEMORY_AND_DISK_SER if the estimated size exceeds the aggregate data memory to prevent unnecessary spilling. Finally, as a cleanup step, we remove unnecessary checkpoints: for example, checkpoints in simple update chains or for small datasets that fit into the driver.

**Repartition Injection:** Operations like join or reduceByKey that cause shuffle are very expensive operations on Spark because shuffle significantly dominates execution time compared to reads from distributed cache. This

Table 1: Physical Spark Matrix Multiply Operators.

| Operator | Pattern | Constraints |
|---|---|---|
| MapMM | $\mathbf{X}\,\mathbf{Y}$ | $\mathcal{M}(\mathbf{X}, \text{out}) < \bar{\mathcal{M}}_{\text{B}}$ |
| | | $\vee\ \mathcal{M}(\mathbf{Y}, \text{out}) < \bar{\mathcal{M}}_{\text{B}}$ |
| MapMMChain | $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$ | $\mathcal{M}(\mathbf{w}) + \mathcal{M}(\mathbf{v}) < \bar{\mathcal{M}}_{\text{B}}$ |
| | | $\wedge\ ncol(\mathbf{X}) \le B_c$ |
| TSMM | $\mathbf{X}^\top\mathbf{X}$ | $ncol(\mathbf{X}) \le B_c$ |
| ZIPMM | $\mathbf{X}^\top\mathbf{Y}$ | $ncol(\mathbf{X}) \le B_c$ |
| | | $\wedge\ ncol(\mathbf{Y}) \le B_c$ |
| CPMM | $\mathbf{X}\,\mathbf{Y}$ | – |
| RMM | $\mathbf{X}\,\mathbf{Y}$ | – |
| PMM | $rmr(diag(\mathbf{v}))\,\mathbf{X}$ | $\mathcal{M}(\mathbf{v}) < \bar{\mathcal{M}}_{\text{B}}$ |

is especially true if shuffle requires spilling. However, Spark avoids unnecessary shuffle if, for example, the inputs to a `join` are partitioned with the same partitioning function because it guarantees co-partitioning but not necessarily co-location. We exploit this optimization feature in Spark with partitioning-preserving operations like `zipmm` that avoids key changes to retain co-partitioning. Our rewrite rule for repartition injection is to introduce explicit repartition operations before loops if the loop contains operations that can potentially exploit the created partitioning in order to avoid *repeated shuffling* of large datasets per iteration.

## 3.2 Memory Budgets and Constraints

As a basis for operator selection and more advanced optimizations, we need to clarify memory estimates and constraints in terms of memory budgets first. A valid execution plan, satisfies these constraints for all operations.

**Memory Budgets:** Given a configuration of Spark driver memory $\mathcal{M}_{\text{D}}$, executor memory $\mathcal{M}_{\text{E}}$, number of executors $|E|$, data fraction $\delta$, and shuffle fraction $\sigma$, we derive SystemML's effective memory budgets as follows. Our control program memory budget $\bar{\mathcal{M}}_{\text{CP}}$ for in-memory single-node operations is given by $\bar{\mathcal{M}}_{\text{CP}} = \alpha\mathcal{M}_{\text{D}}$ (default $\alpha$=0.7). The broadcast memory budget $\bar{\mathcal{M}}_{\text{B}}$ is derived with $\bar{\mathcal{M}}_{\text{B}} = \beta\delta\mathcal{M}_{\text{E}}$ (default $\beta$=0.3) as broadcasts are managed in data space. Similarly, the total data memory is calculated as $|E|\delta\mathcal{M}_{\text{E}}$. Dynamic memory management—introduced in Spark 1.6—increases our memory budgets by removing $\sigma$ and thus, increasing $\delta$ but all constraints still apply.

**Memory Estimates:** SystemML relies on worst-case memory estimates per operation $\mathcal{M}(\text{op})$ [8], reflecting memory requirements of an in-memory single-node operation. This estimate is the sum of memory estimates of all inputs, intermediates and outputs, as each operation pins all its inputs and outputs into memory. $\mathcal{M}(\mathbf{X})$ denotes the memory estimate of a single-block matrix and $\mathcal{M}(\mathbf{X}_{\text{P}})$ denotes the memory estimate of a block partitioned matrix as described in Subsection 4.1. These matrix size estimates are important constraints for operator selection with regard to execution types and broadcast-based operators. All estimates are based on size information, i.e., dimension and sparsity, propagated from the program inputs over the entire program.

**Optimization Objective:** Finally, our optimization objective $\phi$ is to minimize total program execution time subject to memory constraints over all operations and execution contexts (e.g., driver or distributed tasks).

## 3.3 Operator Selection

Similar to SystemML on MapReduce, by default we compile hybrid runtime plans including operators with execution type of in-memory single-node (CP) and operators with execution type Spark. The choice of execution type has high impact on both performance and robustness.

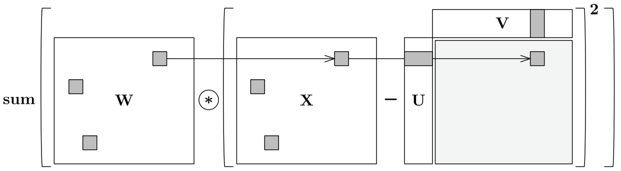


Figure 1: Example Weighted Squared Loss.

**Basic Spark Execution Type:** Apart from more advanced optimizers, SystemML uses a simple heuristic for execution type selection that typically works very well for data-intensive machine learning tasks. Given the memory estimate of an operation $\mathcal{M}(\text{op})$, we schedule op to Spark if $\mathcal{M}(\text{op}) > \bar{\mathcal{M}}_{\text{CP}}$; otherwise to CP.

**Transitive Spark Execution Type:** Exploiting the low latency of Spark jobs, we further transitively pull certain operations into Spark execution type if the main input to an operation op already has the Spark execution type even though $\mathcal{M}(\text{op}) \le \bar{\mathcal{M}}_{\text{CP}}$. For example, consider $sum(\mathbf{X}\mathbf{v})$ and assume that $\mathbf{X}$ is large but the output of $\mathbf{X}\mathbf{v}$ is moderately small. Although we could execute the sum in CP, we execute it in Spark in order to reduce data transfer between executors and the driver. We do this for unary aggregates, unary and matrix-scalar operations, as well as matrix multiplications, but only if the Spark input does not have multiple consumers and is not triggering computation via an action.

**Physical Operator Selection:** Depending on the chosen execution type, we select physical operators according to data and cluster characteristics. For example, consider Spark distributed matrix multiplication. Table 1 shows the physical operators that the optimizer can choose from, based on operation patterns, validity constraints and costs. A common approach is to avoid shuffle by (1) broadcasting one of the two input matrices (e.g., `mapmm`, `mapmmchain`, `pmm`), (2) partitioning-preserving operations (e.g., `zipmm`), or (3) special pseudo-unary operations (e.g., `mapmmchain`, `tsmm`, or `pmm`). If none of these shuffle-avoiding operations apply—for example, due to violated memory or block size constraints—we fall back to `cpmm` or `rmm` [15]. The cost model for this decision on Spark is—similar to the decision for MapReduce—based on estimated shuffle costs weighted by effective parallelism. Similar operator selection decisions are being made for many different operators including matrix-vector binary element-wise operations, right indexing, and more complex fused matrix multiplication operators.

**Fused Physical Operators:** In addition to the basic physical matrix multiplication operators described so far, SystemML also provides a variety of fused operators in order to (1) exploit sparsity via selective computation, (2) reduce the number of intermediate results, and (3) for in-memory single-node computation, to reduce buffer pool evictions. For example, consider the inner loop of our running ALS example, where $\mathbf{W}$ is the non-zero indicator matrix of our sparse input matrix $\mathbf{X}$. Computing the update rules $(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}))\mathbf{V}^\top$ and $\mathbf{U}^\top(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}))$ naïvely would create huge dense intermediates for $\mathbf{U}\,\mathbf{V}$ in the dimensions of $\mathbf{X}$. Similar patterns occur in other factorization algorithms like PNMF and ENMF [22], deep learning, and loss computation such as weighted squared loss $sum(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V} - \mathbf{X})^2)$. Figure 1 shows—by example of computing the weighted squared loss—how we exploit sparsity for selective compu-

**Table 2: Fused Physical Spark Operators.**

| Operator | Example Patterns |
|---|---|
| Map/Red WSLoss | $sum(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}^\top - \mathbf{X})^2)$<br>$sum((\mathbf{X} - \mathbf{W} \odot (\mathbf{U}\,\mathbf{V}^\top))^2)$ |
| Map/Red WSigmoid | $\mathbf{X} \odot sigmoid(\mathbf{U}\,\mathbf{V}^\top)$<br>$\mathbf{X} \odot log(sigmoid(-(\mathbf{U}\,\mathbf{V}^\top)))$ |
| Map/Red WDivMM | $(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}^\top))\mathbf{V}$, $(\mathbf{W}/(\mathbf{U}\,\mathbf{V}^\top))\mathbf{V}$<br>$(\mathbf{U}^\top(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}^\top)))^\top$ |
| Map/Red WCeMM | $sum(\mathbf{X} \odot log(\mathbf{U}\,\mathbf{V}^\top))$ |
| Map/Red WuMM | $\mathbf{X} \odot exp(\mathbf{U}\,\mathbf{V}^\top)$, $\mathbf{X}/(\mathbf{U}\,\mathbf{V}^\top)^2$ |

tation. We use the sparse matrix $\mathbf{W}$ and the sparse-safe multiply $\odot$ as a *sparse driver*, where we only iterate over non-zero entries and selectively compute necessary entries in $\mathbf{U}\,\mathbf{V}$. Due to row-major representation, we support the pattern $sum(\mathbf{W} \odot (\mathbf{U}\,\mathbf{V}^\top - \mathbf{X})^2)$ (with additional transpose if required) for cache-friendly access to $\mathbf{V}$. Table 2 summarizes special fused physical Spark operators along with example patterns for which these operators apply. All of these operators can be executed via broadcasts (map) or join (reduce), where we apply the broadcast-based operators if $\mathcal{M}(\mathbf{U}) + \mathcal{M}(\mathbf{V}) \leq \bar{\mathcal{M}}_\text{B}$, and we have only three inputs or $\mathbf{W} = (\mathbf{X} \neq 0)$; otherwise we fall back to join-based realizations which requires a shuffling of the large matrices (unless co-partitioned) but we still broadcast $\mathbf{U}$ or $\mathbf{V}$ if possible. Our running example applies various instances of `wdivmm` and `wsloss`. Finally, SystemML supports many more fused physical operators for common basic patterns like $sum(\mathbf{X}^2)$.

## 3.4 Extended ParFor Optimizer

Besides data-parallel computation, SystemML also supports task-parallel computation via so-called parallel for (parfor) loops as well as hybrid parallelization strategies that involve both [8]. Examples are descriptive statistics, ensemble learning/cross validation, and algorithms like KMeans, and Multiclass SVM. In addition to the previously described general optimizer extensions, the parfor runtime optimizer requires Spark-specific extensions.

**Physical ParFor Operators:** Our Spark backend supports—similar to our MapReduce backend—three physical operators to execute the entire parfor loop. First, *local* parfor executes multi-threaded workers in the driver which allows for multi-threaded single-node execution and concurrent Spark jobs. Second, *remote* parfor executes the entire loop as a single map-side Spark job. Third, *remote_dp* parfor partitions a given input matrix into disjoint slices and executes the parfor body per slice. There are also separate jobs for data partitioning and result merge. The major difference of remote operators to MapReduce is the memory handling. Since multiple tasks share a common executor process, they also share the same buffer pool which is beneficial for I/O reduction. However, due to separate data and shuffle memory in Spark, there is less memory available ($\alpha(1 - \delta - \sigma)\mathcal{M}_\text{E}$) for computation and intermediate results per task.

**Deferred Checkpoint/Repartition Injection:** In order to allow for all parfor operators, we defer Spark-specific rewrites on caching/checkpointing and repartitioning (see Subsection 3.1) in the parfor body until parfor optimization during runtime. The parfor optimizer then only injects these operators in case of *local* parfor. Checkpoint compilation takes into account the resulting memory budget based on the degree of parallelism. Similarly, we inject repartitioning only if there is a `zipmm` or `cpmm` (due to unknowns) in the body that would benefit from existing partitioning. These rewrites have high impact: for example, the repartitioning creates a `ShuffledRDD` once, avoiding shuffle *per iteration.*

**Eager Checkpointing/Repartitioning:** In case of *local* parfor with repartitioning or pending checkpointing, the parfor optimizer performs eager execution via a simple `count` action, before starting the local workers and with that concurrent Spark jobs. This is beneficial for concurrent Spark jobs that all access a shared but non-cached input RDD because it overcomes thread contention on the block manager and avoids persisting individual partitions multiple times.

**Fair Scheduling for Concurrent Jobs:** For fair scheduling across concurrent Spark jobs, we use by default Spark's fair scheduler via `spark.scheduler.mode=FAIR`. In case of *local* parfor, every worker then setups a thread-local scheduler pool, with `FAIR` scheduling across pools but `FIFO` scheduling within each pool. This improves performance for two major reasons. First, it allows for implicit scan sharing, especially for out-of-core matrices that are evicted from aggregated memory. Second, fair scheduling can increase temporal locality because with FIFO, cached intermediates are more likely to be evicted by other jobs.

**Degree of Parallelism:** The parfor optimizer decides—based on memory estimates and budgets—on the degree of parallelism. To reflect the runtime memory requirements, we extended the parfor memory estimates for Spark. First, distributed RDD operations also require local driver memory if they broadcast inputs. We discuss the broadcast mechanism for partitioned matrices in detail in Subsection 4.1. Second, both `collect` and `parallelize`—as used for the transfer between single-node and distributed operations—require more than twice the memory of read or write. SystemML ensures robustness via so-called *guarded collect* and *parallelize*, which redirect the transfer over HDFS if needed. To avoid unnecessary I/O, the parfor optimizer on Spark uses a more conservative degree of parallelism for *local* parfor by optimizing against a memory constraint of $\bar{\mathcal{M}}_\text{CP}/2$ if the parfor body includes Spark operations.

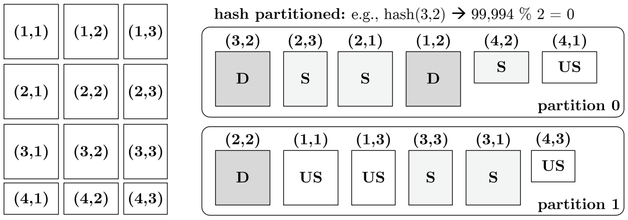
# 4. RUNTIME INTEGRATION

Our Spark backend leverages Spark's Java API and lazy evaluation, which greatly simplifies the runtime integration. In this section, we give an overview of SystemML's Spark backend with regard to distributed matrix representations, hybrid runtime plans via buffer pool integration, implications for dynamic recompilation, partitioning-preserving operations, and specific optimizations.

## 4.1 Distributed Matrix Representation

SystemML supports various text and binary input formats, all of which are internally converted to binary block matrices. Most operations then work on these binary block matrices, which simplifies the compiler and runtime.

**Binary Block Matrices:** Distributed matrices in SystemML are partitioned into fixed size blocks and represented as `JavaPairRDD<MatrixIndexes, MatrixBlock>`, where `MatrixIndexes` are `(long,long)`-pairs of row/column block indexes. Similar structures of tiles, chunks, or blocks, are widely used in existing large-scale ML systems [18, 33, 40]. The fixed block size leads to variable physical sizes but simplifies join processing for binary operations. Figure 2(a) shows an example of a distributed matrix. We use square blocks of size $B_c = 1\text{K}$, which has two benefits. First, the small dimensions lead to a maximum size of 8 MB for dense



(a) Logical (b) Physical Partitioning

**Figure 2: Distributed Matrix Representation.**

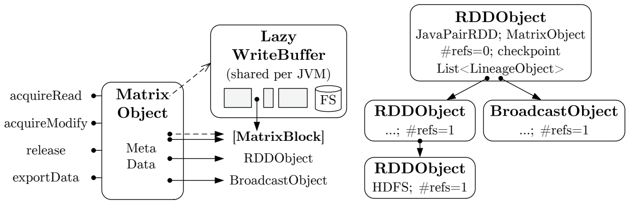
blocks, which is less than common L3 cache sizes. This does not restrict the degree of parallelism and leads to cache-friendly behavior, especially for long chains of pipelined operations. At the same time, these blocks are large enough to amortize per-block overheads. Second, having square blocks also simplifies various operations. For example a transpose $\mathbf{X}^\top$ can be realized by flipping block indexes and a transpose per block, avoiding the need for data shuffle across blocks. In such a distributed block representation, each block is represented in either sparse or dense format, allowing block-local format decisions and robustness with regard to sparsity skew. Sparse matrix blocks use a modified compressed sparse row (MCSR) format that allows efficient incremental construction. On checkpoints, however, we convert these blocks into the more memory-efficient CSR format to avoid unnecessary spilling. We also materialize empty blocks to allow sparse-unsafe operations like `X+7`. However, the compiler injects filters for non-empty blocks in case of chains of sparse-safe operations with consumers like single-node operations. For single-node computation, we represent the entire matrix as a single matrix block to enable reuse and consistency of runtime operations across backends.

**Serialization and Partitioning:** Shuffle is a major bottleneck for SystemML on Spark. We aim to reduce this overhead by (1) minimizing the serialized size of shuffled RDDs, and (2) avoiding shuffle via co-partitioning. First, for small serialized size, we redirect the Java serialization via `externalizable` to our custom block serialization, where we support four formats: empty, dense, sparse, and ultra-sparse. Empty blocks have only a small overhead of 9 B, and dense blocks store values of all matrix cells. Sparse blocks, however, store either rows of column-index/value pairs (sparse) or triples of row-index/column-index/value (ultra-sparse). Space-efficient block formats together with Spark's shuffle compression ensure small transfers. Second, we consistently use the default hash partitioner to (1) achieve good load balance, and (2) avoid breaking DAGs of partitioning-preserving operations (as discussed in Subsection 4.4). Figure 2(b) shows an example of physical RDD partitions with hash partitioning. Only checkpointing and repartitioning change the number of partitions via `coalesce` to the HDFS block size to increase local aggregation.

## 4.2 Buffer Pool Integration

Hybrid runtime plans, composed of single-node operations and distributed Spark operations pose the challenges of exchanging intermediates between runtime backends and robustness with regard to lazy evaluation. In this subsection, we describe the buffer pool integration of Spark operations and related abstractions, which overcome these challenges.

**Basic Buffer Pool Integration:** SystemML executes a compiled program of program blocks and instructions by passing along an execution context that holds the symbol table of currently live matrix objects and scalars. Matrix objects are meta data handles to in-memory or distributed matrices and internally interact with the buffer pool. In memory operations call basic primitives described below. Figure 3(a) shows the integration of RDDs and broadcasts into this architecture. The execution context (that also holds the Spark context), then provides two additional (*) primitives to obtain RDDs and broadcasts for a given variable, where we set `spark.driver.maxResultSize` to unlimited, to enable arbitrarily large transfers between backends.



(a) MatrixObject / WriteBuffer (b) Lineage Tracking

**Figure 3: Buffer Pool Integration.**

- `acquireRead` pins a matrix block into memory. This includes potential read from HDFS or evicted matrices, and triggers pending RDD operations via `collect`.
- `acquireModify` sets and pins a newly created matrix block, which puts the matrix object in state dirty.
- `release` unpins a matrix block, handing new blocks over to a lazy write buffer that handles evictions.
- `exportData` writes a matrix object to HDFS if its state is dirty or the format differs from the existing one.
- `getRDD(*)`: An RDD might be obtained from various states: (1) an existing RDD handle (not cached), (2) a cached matrix block that we would pin, prepare as list of blocks, and `parallelize`, or (2) an HDFS file where we would create the RDD via `hadoopFile`.
- `getBroadcast(*)`: Similar to `getRDD`, broadcasts are either (1) reused broadcast handles, or (2) created from in-memory blocks obtained via `acquireRead`.

**Lineage Tracking:** A major issue with regard to cleanup of variables is lazy RDD evaluation since related RDDs or broadcasts might still be referenced in pending RDD operations. We address this issue by explicit lineage tracking as shown in Figure 3(b). Every Spark instruction that creates an output RDD maintains the lineage to input RDDs and broadcasts. During runtime, we build up a lineage graph of RDD and broadcast objects and their data dependencies. Each lineage object also holds back references to its matrix object and additional meta data such as reference counts $r$. On cleanup, we clear the back reference and check $r$ of the associated lineage objects. In case of $r = 0$, we unpersist the given RDD or broadcast, cleanup associated files on HDFS, decrement $r$ of children, and recursively cleanup child lineage objects until the first object with $r > 0$. This approach ensures robustness with lazily evaluated RDDs and provides access to all data-dependent RDDs, enabling advanced optimizations like short-circuit read/collect and cache handling.

**Guarded Collect/Parallelize:** In contrast to data exchange over HDFS, `collect` or `parallelize` have more than twice the memory requirements of a given matrix because we have to hold the matrix and a collection of partitioned blocks. This would lead to potential out-of-memory situations or unnecessary memory overestimation for all operations. We address this issue by *guarded collect* and *guarded parallelize* where the data transfer is redirected over HDFS if necessary. Exchange over HDFS is significantly faster than `toLocalIterator` because it avoids repeated scans and job latency per partition. The constraint for a plain in-memory `collect` is $\mathcal{M}(\mathbf{X}) + \mathcal{M}(\mathbf{X}_\text{P}) + \mathcal{M}(C) \leq \bar{\mathcal{M}}_\text{CP}$, where $\mathcal{M}(C)$ is the size of currently pinned matrices (tracked below the matrix object interface and in a thread-local manner).

**Partitioned Broadcast Variables:** Broadcasting matrices is crucial for performance because it prevents shuffle and simplifies runtime operators. However, the default broadcast mechanism of `Broadcast<MatrixBlock>` has two major problems. First, it requires function-local partitioning or repeated slicing to enable block-wise operations. Function-local partitioning is not applicable because it increases memory requirements in the number of cores per executor. Second, Spark has a limitation of 2 GB for broadcast variables. We address these issues with *two-level broadcast partitioning* encapsulated in a `PartitionedBroadcastMatrix`. At the first level, we create an array of broadcasts each guaranteed to be smaller than 2 GB. Each broadcast holds a `PartitionedMatrixBlock` consisting of individual matrix blocks with default block size, which is the second level. Partitioning is done at the driver, requiring $\mathcal{M}(\mathbf{X}) + \mathcal{M}(\mathbf{X}_\text{P})$ memory, which is an additional constraint for operator selection. This abstraction enables random access of shared (i,j)-blocks by reference.

## 4.3 Dynamic Recompilation

Dynamic recompilation is applied at natural boundaries between program blocks or artificial recompilation points, created by the optimizer [9]. The goal is to adapt the runtime plan to changing or initially unknown data characteristics.

In contrast to our MapReduce backend, recompilation in the context of Spark poses additional challenges. At the time of recompilation, the size of certain intermediates might still be unknown due to lazy evaluation. We address this by best effort maintenance of output characteristics in all Spark instructions. Many linear algebra operations, element-wise computations, and statistical functions allow to exactly infer the output dimensions for known input characteristics. Several operations like transpose or sort, even preserve all input characteristics. Given the unconditional scope of runtime maintenance, this often allows us to infer—despite initial unknowns—at least the dimensions. For data-dependent operators like `table`, `aggregate`, or `removeEmpty`, we exploit the operation-internally computed output sizes to update the meta data. Finally, we explicitly trigger RDD evaluation if matrix characteristics are still unknown but operations require them for correctness (e.g., `nrow(X)`).

## 4.4 Partitioning-Preserving Operations

Since Spark leverages co-partitioned inputs to prevent shuffle, we explicitly inject repartition operations to avoid repeated shuffle as discussed in Subsection 3.1. In order to fully exploit co-partitioning, we additionally need to (1) preserve partitioning over chains of operations, and (2) exploit partitioning through custom physical operators.

**Partitioning-Preserving Operations:** With hash-partitioning, operations are generally partitioning-preserving if keys do not change; or more precisely, if Spark detects that keys do not change. Unfortunately, even a single operation can lose partitioning and with that break entire DAGs of co-partitioned intermediates into multiple stages. Therefore, we carefully implemented all applicable operations in a partitioning-preserving manner, which follows two general schemes. First, we use more restrictive APIs like `mapValues` (without key access) instead of `mapToPair` for operations like unary, matrix-scalar, or certain unary aggregates (e.g., `rowSums(X)` if $ncol(\mathbf{X}) \leq B_c$). Second, if restrictive APIs are not applicable, we fallback to full partition computation, declared as partitioning-preserving. For example consider a matrix-vector element-wise operation like $\mathbf{X} \odot \mathbf{v}$ with broadcast of $\mathbf{v}$. This operation is partitioning-preserving but needs the input keys for broadcast block lookups. Hence, we use `mapPartitionsToPair(...,true)` to indicate preserved partitioning. We also return lazy iterators instead of materialized outputs per partition to reduce memory requirements.

**Partitioning-Exploiting Operations:** RDD operations can exploit the preserved partitioning to avoid shuffle (implicit or explicit). Implicit partitioning-exploiting operations such as matrix-matrix element-wise operations use primitives like `join` or `cogroup`. Spark automatically exploits partitioning if input and output have the same number of partitions. Hence, we refrain from changing the number of partitions on these operations. In addition, we introduced explicitly partitioning-exploiting operations like `zipmm` for the common matrix multiplication pattern $\mathbf{X}^\top\mathbf{Y}$ (under constraint $ncol(\mathbf{X}) \leq B_c \wedge ncol(\mathbf{Y}) \leq B_c$). The transpose $\mathbf{X}^\top$ would flip block indexes and thus destroy partitioning. A subsequent `cpmm` would then create custom join indexes over the common dimension. Given the block size constraint, however, `zipmm` realizes the matrix multiplication as a zipper-like 1-1 join on the original keys, leveraging existing partitioning. This makes `zipmm` a great fallback when we cannot select the broadcast-based `mapmm` anymore.

**An Example:** To summarize, consider Multiclass SVM (with an one-against-the-rest approach) as an example algorithm, sketched in the following script snippet. Assume characteristics where vectors in $nrow(\mathbf{X})$ neither fit into the driver nor the broadcast budget and $ncol(\mathbf{X}) \leq B_c$.

```
 1: parfor(iter_class in 1:num_classes) {
 2:   Y_local = 2 * (Y == iter_class) - 1
 3:   g_old = t(X) %*% Y_local ...
 4:   while( continue )  {
 5:     Xd = X %*% s
 6:     ... inner while loop (compute step_sz)
 7:     Xw = Xw + step_sz * Xd;
 8:     out = 1 - Y_local * Xw;
 9:     out = (out > 0) * out;
10:     g_new = t(X) %*% (out * Y_local) ...
```

The parfor optimizer injects a repartition of $\mathbf{X}$ because two large matrix multiplications (lines 3, 10) are compiled to `zipmm` that benefits from partitioning. The `mapmm` for line 5 is also partitioning-preserving because $ncol(\mathbf{X}) \leq B_c$, which implicitly propagates over subsequent binary vector operations (lines 6-10). Both inputs for the `zipmm` on line 10 are known to be co-partitioned because `Xd` (line 5) originates from $\mathbf{X}$, preserves partitioning, and is joined back to $\mathbf{X}$.

## 4.5 Specific Runtime Optimizations

Additionally, we introduced the following Spark-specific runtime optimizations to overcome unnecessary overhead.

**Lazy Spark-Context Creation:** On creation of the Spark context, executors are allocated and initialized, which can take up to 20 s. With hybrid runtime plans, this causes large unnecessary overhead if all operations are single-node operations that execute in a few seconds. Hence, we lazily allocate the context on demand when it is first accessed (e.g., when creating an RDD), which avoids context creation for pure single-node computation. Also, for `EXPLAIN` and programs with initial unknowns, we defer the access to the context (e.g., for memory budgets) as long as possible.

**Short-Circuit Read:** Due to conservative checkpoint compilation, a single-node operation might directly consume a cached RDD, which causes unnecessary overhead for reading the matrix into distributed memory and transfering it back to the driver via `collect`. Our *short-circuit read* bypasses the checkpoint based on lineage information and directly reads the matrix from HDFS into the driver.

**Short-Circuit Collect:** Our *short-circuit collect* generalized upon the concept of *short-circuit read*, and bypasses RDD caching, if the given RDD is marked for caching but not yet actually cached. Avoiding caching reduces the overhead of unnecessary reads and memory pressure.

# 5. EXPERIMENTS

We study end-to-end performance characteristics of SystemML on Spark over a variety of ML algorithms and data characteristics as well as specific optimization techniques.

## 5.1 Experimental Setting

**Cluster Setup:** We ran all experiments on a 1+6 node cluster, i.e., one head node of 2x4 Intel E5530 @ 2.40 GHz-2.66 GHz with hyper-threading enabled and 64 GB RAM, as well as 6 nodes of 2x6 Intel E5-2440 @ 2.40 GHz-2.90 GHz with hyper-threading enabled, 96 GB DDR3 RAM @1.33 GHz, 12x2 TB disks, 10Gb Ethernet, and CentOS Linux 7.1. As the runtime environment, we used OpenJDK 1.8.0_65 64bit, Hadoop 2.7.0, Spark 1.5.2, and SystemML as of 02/2016. We ran Spark in `yarn-client` mode from the head node, with 6 executors, 20 GB driver memory, 55 GB executor memory, and 24 cores per executor. For map/reduce tasks, we configured a maximum heap size of 1.6 GB (2 GB container size) and 384 MB sort buffer. Finally, SystemML's memory budget ratio was $\alpha = 0.7$.

**ML Algorithms:** Our experiments cover a variety of common ML algorithms for regression, binomial and multinomial classification, as well as clustering. Table 3 summarizes the used algorithms and parameters (number of iterations, convergence tolerance, regularization, intercept, number of classes/centroids/rank, use of parfor loops), which also indicates essential characteristics such as iterative algorithms, nested loop structures, etc. In detail, we use L2SVM (L2-regularized Support Vector Machines), GLM (Generalized Linear Models, binomial with probit link function), LinregCG (Linear regression with a conjugate gradient method), LinregDS (Linear regression with a direct solve method), MLogreg (Multinomial logistic regression), MSVM (Multiclass L2SVM), Naïve Bayes, KMeans, and ALS. Note that KMeans uses 10 runs all of which with Maxi=20.

**Datasets:** We evaluate the training performance of the previously described algorithms on synthetic datasets. To isolate scalability effects, we keep the number of features (columns) constant at $n = 1{,}000$, and the sparsity (fraction of non-zeros to cells) constant at $sp = 0.9$, but vary the number of rows $m \in \{10^4, 10^5, 10^6, 10^7, 10^8\}$, which corresponds to 80 MB, 800 MB, 8 GB, 80 GB, and 800 GB in dense binary representation. As a shorthand, we call these scenarios XS, S, M, L, and XL. Given the driver memory budget of $\bar{\mathcal{M}}_{\text{CP}} = \alpha \cdot 20\,\text{GB} = 14\,\text{GB}$, we can fit XS, S, and M into the driver. Furthermore, given the total executor data memory of $|E|\delta\mathcal{M}_{\text{E}} = 6 \cdot 0.6 \cdot 55\,\text{GB} = 198\,\text{GB}$, scenario L fits into the distributed cache, whereas XL is an out-of-core scenario. We use these scenarios for all algorithms except ALS because its common data characteristics are largely different and hence, separately introduced in Subsection 5.6.

**Table 3: Characteristics of Used ML Algorithms.**

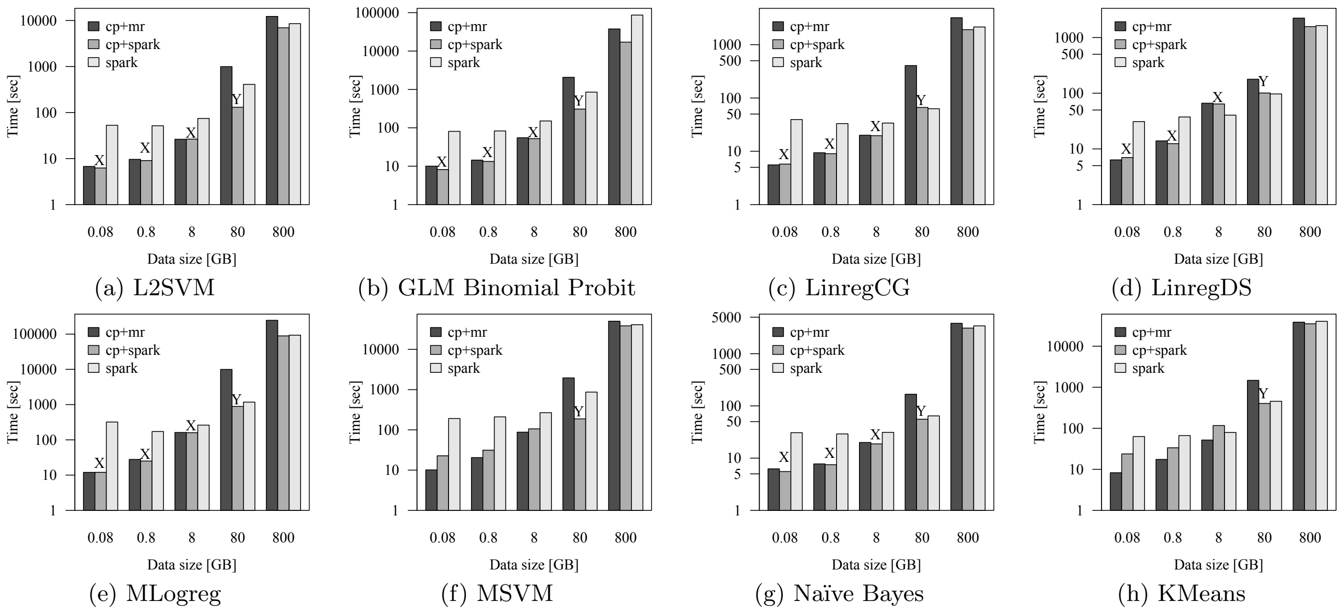
| Algorithm | Maxi | $\epsilon$ | $\lambda$ | Icpt | #C | ParFor |
|---|---|---|---|---|---|---|
| L2SVM | $20/\infty$ | 1e-6 | 1e-2 | N | 2 | N |
| GLM | $20/\infty$ | 1e-6 | 1e-2 | N | 2 | N |
| LinregCG | 20 | 1e-6 | 1e-2 | N | N/A | N |
| LinregDS | N/A | N/A | 1e-2 | N | N/A | N |
| MLogreg | $20/\infty$ | 1e-6 | 1e-2 | N | 5 | N |
| MSVM | $5 \times 20/\infty$ | 1e-6 | 1e-2 | N | 5 | Y |
| Naïve Bayes | N/A | N/A | N/A | N | 5 | Y |
| KMeans | $10 \times 20$ | 1e-4 | N/A | N | 10 | Y |
| ALS | 6/50* | 0 | 1 | N/A | 50 | N |

**Baseline Comparisons:** To understand the characteristics of SystemML on Spark, we compare three different execution modes, namely `cp+mr`, `cp+spark`, and `spark`. By default, SystemML uses the hybrid execution modes, but the full `spark` execution mode—where every matrix operation is executed as a distributed operation—serves as a baseline to evaluate hybrid runtime plans. These execution modes allow for a systematic evaluation because they share the same runtime block operations. We refrain from comparisons to existing machine learning libraries like MLlib [25] due to (1) different abstraction levels of algorithm specification, and (2) different algorithm choices and parameters. A fair comparison would require a benchmark including both accuracy and runtime, which is beyond the scope of this paper.

## 5.2 End-to-End Performance

In the first set of experiments, we investigate the end-to-end performance of our default `cp+spark` execution mode compared to `spark` and `cp+mr` modes, making use of the `spark-submit` and `hadoop` invocation scripts. We report total execution time including invocation overhead, I/O from HDFS and Spark context creation. Figure 4 shows the results for all algorithms and scenarios except ALS.

**Basic Comparison of Spark Execution Modes:** In all cases—except for LinregDS and Kmeans on scenario M —`cp+spark` outperforms full `spark` execution *on average* by 6.1x for relatively small datasets (XS, S, M) and by 1.9x for larger datasets (L and XL). There are three main reasons. First, unlike the `spark` backend, `cp+spark` can exploit lazy Spark context creation and thus, avoids the start-up overhead of up to 20 seconds on small scenarios. The cases when SystemML `cp+spark` backend does not create a SparkContext are annotated with "X" in Figure 4. Note, in these cases, both `cp+spark` and `cp+mr` backends compile identical CP-only plans and hence have similar performance. Second, for scenarios with many operations over small input data and intermediates, in-memory single-node computation is faster than distributed computation on Spark due to multi-threaded implementations (exploiting full single-node parallelism independent of the number of partitions), overhead of distributed operator implementations, and additional job and task latency in the distributed case. For example, con-



(a) L2SVM (b) GLM Binomial Probit (c) LinregCG (d) LinregDS

(e) MLogreg (f) MSVM (g) Naïve Bayes (h) KMeans

**Figure 4: End-to-End Performance of Different Algorithms with Different Execution Modes.**

sider a matrix-vector multiplication on scenario S (800 MB). By exploiting full single-node parallelism, this operation saturates QPI memory bandwidth at 25.6 GB/s, resulting in an execution time of 30 ms. Many pure vector operations even run in sub-milliseconds and hence, even a small Spark job latency of 100 ms creates significant relative overhead. On L and XL scenarios, the improvements are mostly due to the overhead of distributed implementations. Third, lazy evaluation leads to partial repeated execution, whereas in-memory single-node outputs are materialized.

**Compute-Intensive Use Cases:** A counter example to above given reasoning is LinregDS with its core operation $\mathbf{X}^\top\mathbf{X}$, which for 1,000 features is actually compute-intensive such that SystemML's heuristic of execution type selection does not hold on scenario M. For this scenario, `cp+spark` compiles an in-memory `tsmm` whereas for `spark` we run a distributed `tsmm`. Due to the data size of 8 GB, we have 63 HDFS partitions and hence a higher degree of parallelism for distributed computation compared to single-node computation with 16 threads. Accordingly, `spark` outperforms `cp+spark` on scenario M. For larger scenarios, both backends compile a distributed `tsmm` and achieve similar performance.

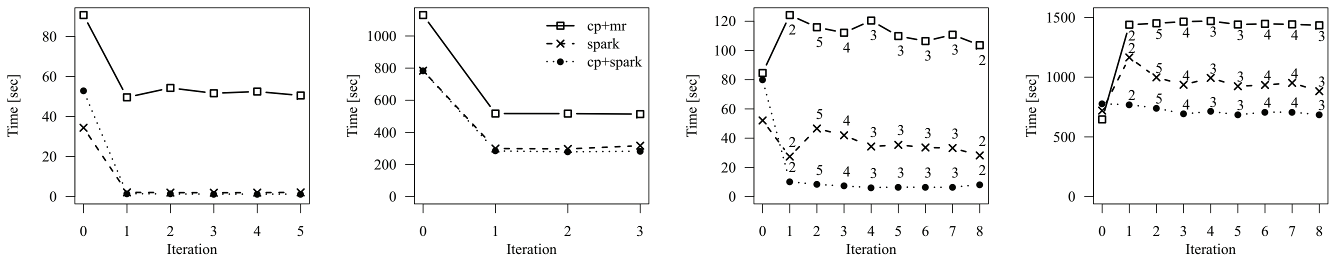
**ParFor Use Cases:** The algorithms MSVM and KMeans exploit parfor loops for task-parallel computation of multiple classes and runs, respectively. Hybrid runtime plans are important for these algorithms because pure CP instructions in the body enable distributed task-parallel computation with *remote* parfor. However, we see that these algorithms do not exploit lazy Spark context creation, even on small input data. This is because the parfor runtime optimizer always probes the Spark context for current cluster characteristics. Both `cp+spark` and `spark` benefit from optimizations like repartitioning injection, eager caching, and fair scheduling. We discuss detailed experiments with regard to these parfor-specific optimizations in Subsection 5.4. Finally, note that `cp+mr` applied runtime piggybacking to batch parallel jobs into fewer jobs for scan sharing in scenarios L and XL.

**Comparison of MR Execution Mode:** In comparison to our `cp+mr` backend, the `cp+spark` backend is 5-10x faster for scenario L (i.e., when the input data fits into aggregated cluster memory but not a single node). This speedup is due to (1) automatic checkpoint injection by SystemML that avoids repeated HDFS reads, and (2) significantly smaller latency of Spark jobs compared to MR. The use cases where this applies are annotated with "Y" in Figure 4. In these cases, `cp+mr` repeatably reads from HDFS, whereas on Spark we only have a single read from HDFS and subsequent reads from memory. On the 800 GB datasets—where the data does not fit into aggregated memory—`cp+spark` is still 1.1 to 2.8x faster than `cp+mr`. The slight performance improvement is due to faster task scheduling with standing Spark executors and because SystemML uses the storage level `MEMORY_AND_DISK` for injected RDD checkpoints, which allows for read from local FS after initial read from HDFS. However, we also observed that an aggressive injection policy of RDD checkpointing can result in performance degradation due to frequent evictions from Spark's in-memory cache to disk. Also note that `cp+mr` sometimes compiles a different parfor plan due to a larger task memory budget.

## 5.3 Runtime per Iteration

We further investigate the runtime per iteration of LinregCG and L2SVM. Figure 5 shows the runtime per iteration for in-memory (80 GB) and out-of-core (800 GB) datasets, where iteration 0 refers to all operations before the outer loop. For both algorithms, iteration 0 covers initial read, triggered by a matrix-vector multiplication $\mathbf{X}^\top\mathbf{y}$. Unlike LinregCG, L2SVM has a nested while loop, where the outer loop minimizes the objective using conjugate gradient steps and the inner loop optimizes the step size using Newton's method in the direction of the gradient. The L2SVM plots are annotated with the number of inner iterations.

**In-Memory Scenario (L, 80 GB):** Due to two passes over $\mathbf{X}$ per outer iteration for L2SVM, `cp+mr` shows higher execution time for iterations 1-8 than for iteration 0. In contrast, RDD caching significantly reduces the time per iteration for `cp+spark` and `spark`, after initial read in iteration 0. This leads to significant improvements of both Spark



(a) LinregCG 80GB (b) LinregCG 800GB (c) L2SVM 80GB (d) L2SVM 800GB

**Figure 5: Runtime per Iteration of LinregCG and L2SVM with Different Execution Modes.**

**Table 4: ParFor-Specific Optimizations for MSVM.**

| Scenario | Runtime | `tak+*` | `zipmm` |
|---|---|---|---|
| All optimizations | 1,470 s | 3,142 s | 564 s |
| Without eager caching | 1,885 s | 4,081 s | 540 s |
| Without fair scheduling | 2,088 s | 3,974 s | 378 s |
| Without repartitioning | 9,102 s | 17,030 s | 18,322 s |

execution modes compared to `cp+mr`, which carries over to end-to-end improvements as the number of iterations until convergence increases. Iteration 0 is more expensive for `cp+spark` due to lazy Spark context creation on the first Spark instruction, whereas for `spark` it is created during compilation. On L2SVM, `cp+spark` also performs significantly better than `spark` because SystemML compiled pure in-memory, single-node operations for the entire inner loop. In contrast, for LinregCG, the plans were almost identical except for very small vector operations in the number of features and hence there are no major differences.

**Out-of-Core Scenario (XL, 800 GB):** For the XL scenario, we see that every iteration requires reads of $\mathbf{X}$ and hence the differences of `cp+mr` and `cp+spark` reduce to around 2x. The small improvements for `cp+spark` are again due to partial read from memory and local disk, as well as smaller Spark job and task latency. On L2SVM, we see that even for large datasets, hybrid runtime plans matter because in-memory single-node operations avoid unnecessary overhead and reduce memory pressure. For LinregCG, we have only a single pass over $\mathbf{X}$ per iteration and hence see that the first read from HDFS is significantly slower than subsequent reads. With regard to such out-of-core scenarios, all backends benefit from special physical operations like `mapmmchain` avoiding an unnecessary pass over $\mathbf{X}$.

## 5.4 ParFor-Specific Optimizations

We ran additional experiments to analyze the impact of various parfor optimizations and partitioning-preserving operations. We used MSVM with a modified L scenario of 250M × 40 dimensions (80 GB) as well as a driver memory of 5 GB, which leads to violated memory budget constraints of `mapmm` and hence a fall-back to the join-based `zipmm` matrix multiplications. Furthermore, any operators over vectors in $nrow(\mathbf{X})$ are now compiled to distributed Spark operations as well. In this scenario, the instructions `zipmm` and `tak+*` (for $sum(\mathbf{v1} \cdot \mathbf{v2} \cdot \mathbf{v3})$, used twice in the inner loop) constitute actions and hence trigger computation. The DML script for MSVM was already discussed in Subsection 4.4.

Table 4 shows the results with individual optimizations disabled and all optimizations enabled. We report the total elapsed runtime and the total execution time of `zipmm` and `tak+*`, where the total elapsed time can be smaller than individual operation runtimes due to a parfor degree of parallelism of 5. We see huge impact of repartitioning because—together with partitioning-preserving and exploiting operations—we avoid shuffle per iteration. Both eager caching and fair scheduling give additional benefits due to avoided thread contention and better locality.

**Table 5: Sparse Matrix Formats for LinRegCG.**

| Scenario | CSR | MCSR Ser. | MCSR |
|---|---|---|---|
| $sp = 0.002$ (5M × 1000K) | 349 s | 764 s | 2,152 s |
| $sp = 0.02$ (5M × 100K) | 126 s | 159 s | 165 s |

## 5.5 Memory-Efficient Sparse Matrices

We further study the effect of converting sparse matrices into more memory-efficient representations on checkpoints as described in Subsection 3.1. In detail, we compare MCSR (without serialization) as a baseline, as well as MCSR (with serialization, i.e, storage level `MEMORY_AND_DISK_SER`) and CSR. We run LinregCG with the same parameters as the end-to-end experiments over synthetic datasets (110 GB) with varying sparsity and number of columns. Table 5 shows the results. Checkpointing the sparse matrix blocks in the memory-efficient CSR format allows SystemML to efficiently utilize Spark's in-memory cache and avoid unnecessary spilling or deserialization overhead. Hence, we observe 2.2x speedup for sparsity of $sp = 0.002$ and 1.3x speedup for sparsity of $sp = 0.02$ compared to MCSR with serialization. Also, we observe that the overhead of deserialized MCSR format (in particular with few non-zeros per block row, i.e., $sp = 0.002$) leads to a slowdown of up to 6.2x.

## 5.6 ALS End-to-End Experiments

Finally, we compare the end-to-end performance of our running ALS example using different execution modes: `cp+mr`, `cp+spark`, and `spark`, in terms of efficiency and scalability on synthetic datasets of varying sizes. To reduce the number of iterations of the inner loop, we used a heuristic to dynamically change the tolerance threshold of the inner loop `tt` by exponential decay: starting from an initial value `t_0=0.1`, we halved parameter `tt` in every outer iteration.

**Datasets:** To reflect common data characteristics, we keep the number of columns (representing items) constant at $n = 10^5$, and the sparsity constant at $sp = 0.01$, but vary the number of rows (representing users) $m \in \{10^5, 10^6, 10^7\}$, which correspond roughly to 1.2 GB, 12 GB, and 120 GB in sparse binary representation. We refer to these datasets as S, M, and L, respectively. We generated the datasets by first creating two rank-50 matrices $\mathbf{L}^*_{m\times 50}$ and $\mathbf{R}^*_{50\times n}$ with entries sampled independently from Normal(0,1) distribution and then taking the absolute value to ensure non-negativity. We obtained the data matrix by sampling $N = sp \cdot m \cdot n$ random entries from $\mathbf{L}^*\mathbf{R}^*$ and adding Normal(0,0.1) noise.

**Table 6: ALS End-to-End Performance.**

| Scenario | cp+mr | cp+spark | spark |
|---|---|---|---|
| S ($10^5 \times 10^5$, 0.01, 1.2 GB) | **131** s | 136 s | 135 s |
| M ($10^6 \times 10^5$, 0.01, 12 GB) | 1,088 s | **342** s | 432 s |
| L ($10^7 \times 10^5$, 0.01, 120 GB) | >24 h | **10,537** s | 15,487 s |

**Setup:** We used the same configuration as described in Section 5.1 across all execution modes, except for a driver memory of 15 GB for the S and M dataset and 30 GB for the L dataset. In our experiments, we compute a rank $r = 50$ factorization by minimizing the L2 regularized squared loss with the regularization parameter $\lambda = 1.0$ throughout. In each case, we ran the ALS algorithm for 6 iterations.

**Comparison Results:** Table 6 summarizes the results of the end-to-end runtime comparison (including the time to read the input matrix, fitting the model, and writing the computed factor matrices) of ALS for all three execution modes. We see that running ALS on `cp+spark` is clearly beneficial across all datasets. On the S dataset, all three execution modes perform similar, even though ALS with `cp+mr` or `cp+spark` run purely in CP mode (multi-threaded on the head node), whereas ALS with `spark` utilizes all the cluster nodes. For the larger M and L scenarios, however, the `cp+spark` generates hybrid execution plans including CP and Spark instructions and outperforms the other execution modes. ALS in `cp+mr` did not terminate within 24 h on dataset L, since the user-factors (matrix `U` in our running example from Subsection 2.2) did not fit into the memory of map/reduce tasks. Consequently, replication and shuffling led to repeated spilling. In contrast, both `spark` and `cp+spark` execution modes generated broadcast-based operators as `U` is shared across executor cores.

## 6. DISCUSSION

Finally, we share major lessons learned with regard to SystemML on top of Spark, but also declarative ML in general.

**Spark over Custom Framework:** SystemML on MR was motivated by sharing cluster resources with other MR-based systems. With YARN things changed as it enables multi-tenancy across frameworks. Before deciding for a Spark backend, we discussed alternatives including a custom framework implemented from scratch. Eventually we have chosen Spark, not just because it is a well-engineered framework with strong contributor base, but mostly to provide users the flexibility of seamless data preparation and feature engineering, which is invaluable for end-to-end pipelines.

**Stateful Distributed Caching:** The ability to exploit distributed caching (of deserialized objects) via standing executors together with Spark's fast task scheduling really made a performance difference for SystemML. However, it also came with challenges like (1) reduced working memory for task-parallel computation, (2) state-dependent memory constraints, (3) global effects of update-in-place, and (4) fair resource management in shared clusters.

**Memory-Efficiency:** In contrast to SystemML on MR, where we process one block at-a-time, memory-efficiency is far more important on Spark to avoid unnecessary cache spilling. Examples of how to reduce memory pressure are custom serialization via `externalizable`, compact data structures for sparse, read-only matrices (e.g., CSR format), and lazy iterators for partition-wise execution.

**Lazy RDD Evaluation:** Some major advantages but also challenges of SystemML's Spark backend were related to lazy evaluation. First, lazy evaluation removed the need for custom piggybacking, i.e., grouping distributed operations into jobs for scan sharing. Job execution based on actions usually works very well, except special cases where (1) separate actions trigger multiple passes over a shared input, or (2) complex DAG structures cause repeated execution of compute-intensive operations. Second, lazy evaluation also made the execution of compiled runtime plans more difficult. Examples are variable cleanup, driver memory management, statistics profiling, runtime plan cost estimation, and dynamic recompilation. Third, lazy evaluation allows to exploit meta data information of inputs such as existing partitioning, which was highly beneficial for SystemML.

**Declarative ML:** Creating the Spark backend also made a great case for declarative ML in general. Data independence of ML algorithms allowed us to leverage RDDs and related operations without changing a single algorithm. Similarly, we were able to automatically exploit distributed caching and partitioning by newly introduced Spark-specific rewrites. The separation of concerns between ML algorithm semantics as well as underlying data structures and execution plan generation also ensures independence of runtime frameworks like MapReduce or Spark. This independence allowed us to adapt to new technology like Spark yet maintaining support for MapReduce v1, which overall protects investments in created custom ML algorithms.

## 7. RELATED WORK

We review related work with regard to alternative specifications and implementations of large-scale ML algorithms.

**Low-Level Primitives:** Beside data-parallel frameworks like MapReduce [14], Spark [41], or Flink [2], there exist frameworks that provide low-level distributed primitives for ML algorithms. For instance, R's *rmr* [27] package exposes map and reduce primitives, HP's Distributed R [37] package supports operations on distributed arrays, and REEF [38] provides a scheduler framework designed for ML-specific iterative task life-cycle management on YARN. Furthermore, graph processing systems like GraphLab [23] often provide vertex-centric abstractions for graph-parallel operations. Users of these frameworks are burdened with the tasks of devising distributed runtime plans, and optimizing them for performance and scalability.

**ML Libraries:** A common approach to large-scale ML is to provide pre-canned distributed implementations of selected ML algorithms as libraries. Examples include Mahout [4], Spark MLlib [25], MADlib [17], Vowpal Wabbit [21], Revolution R ScaleR [28], SkyTree ML software [29], and H2O [16]. Such fixed implementations do not fit all scenarios, cannot be adapted to changing data properties, and do not allow for algorithm customizations without modifying the distributed algorithm implementation.

**ML UDF-Centric Systems:** At a slightly higher abstraction level, there are frameworks that provide users with building blocks and UDF support to construct ML models. Examples are TensorFlow [1] that executes data flow graphs of black-box kernels, DMTK [26] that provides a parameter-server-based framework, MLI [31], MLPACK [13], Shogun [30], Tupleware [12], and Emma [3]. These systems, however, only provide limited automatic optimization of runtime plans and/or data independence.

**Declarative ML:** Systems for declarative ML can be classified into *declarative ML algorithms* and *declarative ML* *tasks* [10]. Example systems aiming for declarative ML algorithms are OptiML [34], SystemML [15], SimSQL [11], SciDB [33], Cumulon [18], DMac [39], and Mahout Samsara [24]. These systems provide a simple yet flexible specification of ML algorithms, and some of them data independence, and automatic optimization and parallelization. Similar to SystemML, DMac and Mahout Samsara also leverage Spark for distributed operations. Frameworks like MLBase [20, 32] or Columbus [42] further enable declarative ML tasks of model or feature selection, where they optimize both model accuracy and performance of ML algorithms.

## 8. CONCLUSIONS

In this paper, we have presented an up-to-date overview of SystemML, necessary engine extensions to deeply exploit Spark, and solutions to unique implementation challenges on Spark such as memory handling and lazy evaluation. Spark-specific key optimizations are automatic injection of RDD caching/checkpointing and repartitioning, as well as partitioning-preserving operations to minimize the number of data scans and shuffles. We use fair scheduling of concurrent Spark jobs issued from multiple parfor threads. Our Spark backend leverages Spark's Java API and lazy evaluation. We also explained our distributed matrix representation in RDDs, explicit lineage tracking for robust cleanup of broadcast and RDD variables, explicit triggering of RDD evaluations for eager caching and repartitioning, dynamic recompilation as well as careful implementation of partitioning-preserving and exploiting operations. Our experiments show that hybrid execution plans are crucial for performance, and when compared to MR backend, our Spark backend provides up to 5-10x speedup when the data fits in memory and up to 2x improvement for datasets larger than the aggregated memory. We also want to note that the rich Spark API significantly simplified the backend implementation for Spark compared to MR. SystemML is open source, and we welcome community contributions.

**Acknowledgments:** We thank Nakul Jindal, Christian R. Kadner, Jihyoung Kim, Narine Kokhlikyan, Deepak Kumar, Min Li, Luciano Resende, Alok Singh, Glenn Weidner, and Wen Pei Yu for their significant contributions to SystemML.

## 9. REFERENCES

[1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, abs/1603.04467, 2016.

[2] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB J.*, 23(6), 2014.

[3] A. Alexandrov et al. Implicit Parallelism through Deep Language Embedding. In *SIGMOD*, 2015.

[4] Apache. *Mahout*, 2016. `http://mahout.apache.org`.

[5] Apache. *Spark*, 2016. `http://spark.apache.org`.

[6] Apache. *SystemML (incubating)*, 2016. `http://systemml.apache.org`.

[7] A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, 2015.

[8] M. Boehm et al. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7), 2014.

[9] M. Boehm et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3), 2014.

[10] M. Boehm et al. Declarative Machine Learning – A Classification of Basic Properties and Types. *CoRR*, abs/1605.05826, 2016.

[11] Z. Cai et al. Simulation of Database-Valued Markov Chains Using SimSQL. In *SIGMOD*, 2013.

[12] A. Crotty et al. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12), 2015.

[13] R. R. Curtin et al. MLPACK: A Scalable C++ Machine Learning Library. *JMLR*, 14(1), 2013.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[15] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[16] H2O. *H2O*. `http://h2o.ai/product/`.

[17] J. M. Hellerstein et al. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB*, 5(12), 2012.

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

[19] B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.

[20] T. Kraska et al. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[21] J. Langford, L. Li, and A. Strehl. Vowpal Wabbit Online Learning Project, 2007.

[22] C. Liu et al. Distributed Nonnegative Matrix Factorization for Web-Scale Dyadic Data Analysis on MapReduce. In *WWW*, 2010.

[23] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 5(8), 2012.

[24] D. Lyubimov. *Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines*. Apache, 2016.

[25] X. Meng et al. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[26] Microsoft. *Distributed Machine Learning Toolkit*.

[27] R-project. *CRAN Task View: High-Performance and Parallel Computing with R*, 2016.

[28] Revolution Analytics. *Revolution R Enterprise ScaleR*, 2016.

[29] Skytree. *Skytree Machine Learning Software*.

[30] Sonnenburg et al. The SHOGUN Machine Learning Toolbox. *JMLR*, 11, 2010.

[31] E. R. Sparks et al. MLI: An API for Distributed Machine Learning. In *ICDM*, 2013.

[32] E. R. Sparks et al. Automating Model Search for Large Scale Machine Learning. In *SOCC*, 2015.

[33] M. Stonebraker et al. The Architecture of SciDB. In *SSDBM*, 2011.

[34] A. K. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*, 2011.

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

[36] V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SOCC*, 2013.

[37] S. Venkataraman et al. Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices. In *Eurosys*, 2013.

[38] M. Weimer et. al. REEF: Retainable Evaluator Execution Framework. In *SIGMOD*, 2015.

[39] L. Yu, Y. Shao, and B. Cui. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, 2015.

[40] R. B. Zadeh et al. linalg: Matrix Computations in Apache Spark. *CoRR*, abs/1509.02256, 2015.

[41] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[42] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.

[43] Y. Zhou et al. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM*, 2008.