SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle

Matthias Boehm[1,2], Iulian Antonov[2], Sebastian Baunsgaard[1]*, Mark Dokter[2], Robert Ginthör[2], Kevin Innerebner[1], Florijan Klezin[2], Stefanie Lindstaedt[1,2], Arnab Phani[1], Benjamin Rath[1], Berthold Reinwald[3], Shafaq Siddiqi[1], Sebastian Benjamin Wrede[2]*

[1] Graz University of Technology; Graz, Austria
[2] Know-Center GmbH; Graz, Austria
[3] IBM Research – Almaden; San Jose, CA, USA

ABSTRACT

Machine learning (ML) applications become increasingly common in many domains. ML systems to execute these workloads include numerical computing frameworks and libraries, ML algorithm libraries, and specialized systems for deep neural networks and distributed ML. These systems focus primarily on efficient model training and scoring. However, the data science process is exploratory, and deals with underspecified objectives and a wide variety of heterogeneous data sources. Therefore, additional tools are employed for data engineering and debugging, which requires boundary crossing, unnecessary manual effort, and lacks optimization across the lifecycle. In this paper, we introduce SystemDS, an open source ML system for the end-to-end data science lifecycle from data integration, cleaning, and preparation, over local, distributed, and federated ML model training, to debugging and serving. To this end, we aim to provide a stack of declarative language abstractions for the different lifecycle tasks, and users with different expertise. We describe the overall system architecture, explain major design decisions (motivated by lessons learned from Apache SystemML), and discuss key features and research directions. Finally, we provide preliminary results that show the potential of end-to-end lifecycle optimization.

1. INTRODUCTION

Machine learning (ML) applications profoundly transform our lives, and many domains such as health care, finance, media, transportation, production, and information technology itself. Increased digitalization, sensor-equipped vehicles and production pipelines, feedback loops in data-driven products, and data augmentation also provide large, labeled data collections for training the underlying ML models.

Existing ML Systems: ML systems to execute these workloads are—due to a variety of ML algorithms and lack

_____
*Work done while at IT University of Copenhagen, Denmark.

of standards—still diverse and rapidly evolving. Major system categories include numerical computing frameworks like R, Python NumPy [72], or Julia [5], algorithm libraries like Scikit-learn [57] or Spark MLlib [48], large-scale linear algebra systems like Apache SystemML [8] or Mahout Samsara [61], and specialized deep neural network (DNN) frameworks like TensorFlow [1], MXNext [13], or PyTorch [55, 56]. These systems primarily rely on numeric matrices or tensors, and focus on efficient ML training and scoring.

Exploratory Data-Science Lifecycle: In contrast to classical ML problems, the typical data science process is exploratory. Stakeholders pose open-ended problems with underspecified objectives that allow different analytics, and can leverage a wide variety of heterogeneous data sources [58]. Data scientists then investigate hypotheses, integrate the necessary data, run different analytics, and look for interesting patterns or models [16]. Since the added value is unknown in advance, little investment is made into systematic data acquisition, and preparation. This lack of infrastructure results in redundancy of manual efforts and computation, especially in small or medium-sized enterprises, which often lack curated catalogs of data and artifacts.

Data Preparation Problem: It is widely recognized that data scientists spend 80-90% of their time finding relevant datasets, and performing data integration, cleaning, and preparation tasks [70]. For this reason, many industrial-strength ML applications have dedicated subsystems for data collection, verification, and feature extraction [3, 62, 64]. Since data integration and cleaning are, however, stubbornly difficult tasks to automate [4], existing work primarily focuses on well-defined subproblems or—like Wrangler [37, 59] and Trifacta [29]—on semi-manual data wrangling through interactive UIs. Unfortunately, this diversity of tools and specialized algorithms lacks broad systems support, requires boundary crossing, and lacks optimization across the lifecycle. These problems motivated various in-database ML toolkits [14, 22, 31, 46, 54, 66] to enable data preparation and ML training/scoring in SQL. However, this approach was—except for success stories like factorized learning [41, 50, 63]—mostly unsuccessful because data scientists perceived in-database ML and array databases [69] as unnatural and cumbersome due to the need for data loading, and the verbosity of linear algebra in SQL [66].

A Case for Declarative Data Science: From the viewpoint of a data scientist, it seems most natural to specify data science lifecycle tasks through familiar R or Python

syntax and use stateless systems, which directly process files or in-memory objects. A key observation is that state-of-the-art data integration algorithms (e.g., for data extraction, schema alignment, entity linking, and data fusion) are themselves based on machine learning [20]. Similar observations can be made for data cleaning [30, 60], outlier detection [12, 32], missing value imputation [11, 71], semantic type detection [35, 81], data augmentation [17], feature selection [74], model selection and hyper-parameter tuning (e.g., via Bayesian Optimization) [24, 39, 51], and model debugging [15, 25]. We aim to leverage this characteristic by extending ML systems with high-level abstractions for the entire data science lifecycle but implement these abstractions with a domain-specific language (DSL) used for ML training and scoring. As a "byproduct", we avoid boundary crossing and the system can perform optimizations across lifecycle tasks.

Contributions: Following this goal of better systems support for declarative data science pipelines, we introduce SystemDS[1], an open source ML system for the end-to-end data science lifecycle from data integration, cleaning, and preparation, over efficient ML model training, to debugging and serving. Our detailed contributions are:

- *Lessons Learned and Vision:* We first reflect on lessons learned from building Apache SystemML (as the predecessor of SystemDS), open problems, and how they influenced the overall vision of SystemDS in Section 2.
- *System Architecture:* Following the outlined vision, we then describe the resulting system architecture and design decisions regarding language abstractions, compilation and runtime backends, as well as the underlying data model of heterogeneous tensors in Section 3.
- *Key Features and Directions:* Subsequently, we discuss key features like lineage tracing, data preparation primitives, and federated ML in Section 4.
- *Preliminary Results:* Finally, we present preliminary results comparing performance with TensorFlow and Julia, and showing optimization opportunities—such as reuse—across lifecycle tasks in Section 5.

2. LESSONS LEARNED AND VISION

Our central goal is to provide high-level abstractions and dedicated system support for the entire data science lifecycle, with a special focus on ML pipelines. Existing end-to-end ML frameworks like TFX [3], KeystoneML [68], or Alpine Meadow [65] are built on top of ML libraries, which allows reusing these evolving systems, but consequently view ML algorithms as black boxes. Cross-library compilation in Weld [52] focuses primarily on UDFs. In contrast, we believe that control of the compiler and runtime is of utmost importance for seamless interoperability and performing optimizations such as fine-grained redundancy elimination. For this reason, we forked SystemDS from Apache SystemML [7, 8, 27] and we are currently rebuilding its foundations.

2.1 Lessons Learned from SystemML

SystemML has been under active development—with fluctuating team size—for about a decade. Here, we share selected lessons learned that influenced the vision, design, and system architecture of SystemDS:

- *L1 Data Independence & Logical Operations:* Physical data independence and high-level linear algebra operations provided great independence of the evolving technology stack (e.g., MR→Spark, and GPUs), simplified development (e.g., library algorithms) and deployment (e.g., large-scale/embedded), and enabled adaptation to changing cluster and data characteristics (e.g., local/distributed, and dense/sparse/compressed).
- *L2 User Categories:* SystemML focused on linear algebra programs for algorithm developers and ML researchers who write new or customize existing ML algorithms. However, this area is a niche as most data scientists work with existing algorithms, but need better support for other lifecycle tasks instead.
- *L3 Diversity of ML Algorithms & Apps:* Today's ML systems literature largely focuses on DNNs, mini-batch SGD, and parameter servers. In practice, however, there is a wide variety of existing ML algorithms (e.g., unsupervised, (semi-)supervised batch $1^{st}/2^{nd}$-order, ensembles, mini-batch DNNs, hybrid batch)—which require very different parallelization strategies—as well as complex ML applications that combine ML algorithms, numerical computing, and rules.
- *L4 Heterogeneous & Structured Data:* SystemML supports feature transformations on frames (2D-tables with a schema). However, many applications deal with heterogeneous data (e.g., multi-modal), various forms of structure, and a wide variety of data corruptions. Boundary crossing for integrating, and preparing these datasets is still a major issue.

Discussion: A natural question is why SystemML ultimately did not—except for few IBM products and services—reach adoption in practice. There are a number of overlapping reasons. First of all, the focus on ML researchers who directly experiment with large data was a niche (*L2*). Organizations that deal with large, distributed datasets often have dedicated teams or use existing libraries. Over the last years, the ML research focus also moved toward mini-batch DNN workloads, parameter servers, and almost exclusively Python bindings (*L3*). Together, these developments rendered SystemML's key differentiator—of automatically compiling R-like scripts into hybrid runtime plans of local and distributed Spark operations—ineffective in spurring adoption. Lacking a pressing need, users gravitated toward more popular frameworks like Scikit-learn, Spark MLlib, PyTorch, and TensorFlow. Although SystemML's R-like (and later Python-like) syntax was chosen to simplify adoption, it was still a DSL, with all its challenges like limited documentation and online resources, as well as difficulties of building an optimizing compiler, especially for unknown workloads.

2.2 SystemDS Vision and Design

Vision: In contrast to traditional ML training and scoring, there is a dire need for more effective data integration, cleaning, and preparation as well as model debugging, especially for large-scale problems. Accordingly, the central goal of SystemDS is to provide high-level abstractions and systems support for the wealth of data science lifecycle tasks (*L3* and *L4*) and users with different expertise (*L2*). Our overall vision comprises three components. First, we aim to implement a hierarchy of abstractions for data science tasks based on a *DSL for ML training and scoring* (Sec-

---

[1]The source code and releases (SystemDS 0.1 published 08/2019) are available at https://github.com/tugraz-isds/systemds.
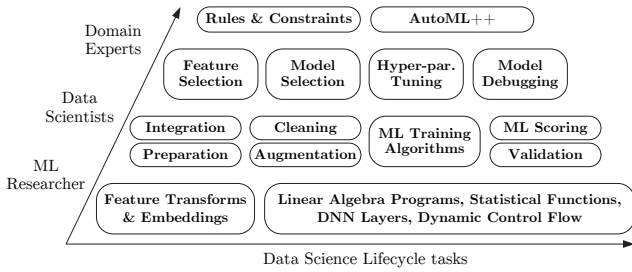
Figure 1: A Stack of Declarative Languages.

tion 3.1) because state-of-the-art data integration, preparation, and cleaning heavily rely on machine learning; because exploratory data science interleaves data preparation, ML training, scoring, and debugging in an iterative process; and because once these tasks are expressed in dense or sparse linear algebra, we expect very good performance. Second, we aim to provide a holistic system infrastructure for the different lifecycle tasks and algorithm classes that require different parallelization strategies. The key to accomplish that are *complementary runtime backends and an optimizing compiler* (Section 3.2). We take advantage of the now relatively mature SystemML compiler and extend it for new architectures like federated learning. Furthermore, the hierarchy of language abstractions inevitably creates fine-grained redundancy, which we aim to eliminate via automatic optimization at compiler and runtime level. Third, supporting data integration and preparation in linear algebra programs requires a more general data model for handling heterogeneous and structured data. In contrast to existing ML systems, our central data model are *heterogeneous tensors* (Section 3.3), i.e., multi-dimensional arrays of different data types, including JSON strings to represent nested data.

Summary: Together, we believe that a holistic system infrastructure for effective and efficient data preparation, ML training and debugging (e.g., distributed data cleaning under awareness of the entire pipeline)—something that cannot be composed from existing libraries—addresses a pressing issue. At the same time, there are lots of open challenges that require novel techniques throughout the system stack.

3. SYSTEM ARCHITECTURE

The outlined vision directly influenced the design and architecture of SystemDS. We now provide an overview of language abstractions, runtime backends, and the underlying data model, before describing selected key features.

3.1 Language Abstractions and APIs

Scripting Language: Following the success of declarative ML (*L1*), we leverage SystemML's DML (Declarative ML Language) [27], a scripting language with R-like syntax for linear algebra, aggregations, element-wise and statistical operations, control flow programs, and user-defined functions. However, we extend this language—as shown in Figure 1—by a stack of declarative abstractions for different lifecycle tasks, and users with different expertise (*L2*). We aim to provide data scientists and domain experts with abstractions for data integration and extraction, cleaning and preparation, data augmentation, model validation, model selection, hyper-parameter tuning, model debugging, rules and AutoML with domain-specific extensions (e.g., constraints and simulation models). To facilitate the development and
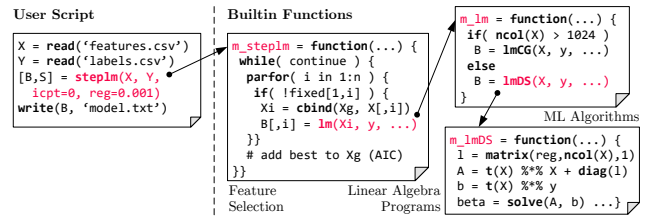

Figure 2: Example Stepwise Linear Regression.

compilation of these abstractions, we introduced a mechanism for registering DML-bodied built-in functions, and we aim to advance existing size propagation techniques.

EXAMPLE 1 (STEPWISE LINEAR REGRESSION). *To see how powerful these abstractions are, consider stepwise linear regression [74], a classical forward feature selection method. This method iteratively runs what-if scenarios and greedily selects the next best feature until the Akaike information criterion (AIC) does not improve anymore (see* `steplm` *in Figure 2). Each configuration trains a regression model via* `lm`, *which in turn calls iterative or closed form linear algebra programs. For an input matrix* $\mathbf{X}$ *(e.g., with* $ncol(\mathbf{X}) = 500$ *features), the compiler then collapses these abstractions—by removing unnecessary branches, dead code elimination, and function inlining—compiles distributed operations if necessary, and can reason about the end-to-end computation.*

APIs and Language Bindings (Fig. 3-1)**:** The user-defined scripts can then be executed with different APIs as shown in Figure 3, where gray-shaded boxes indicate major new components. This includes command line invocation (e.g., through `java` or `spark-submit`) and the programmatic APIs (`MLContext` or `JMLC`). The `MLContext` API allows Spark RDDs and Datasets as script inputs, while `JMLC` is an API for embedded, low-latency scoring that allows pre-compiling a script and repeatedly executing it with different in-memory inputs. For a seamless integration with typical data science workflows, we will further add host language bindings for Python, R, and Java. These bindings expose individual operations, internally collect larger DAGs of operations and entire programs, and finally compile and execute efficient runtime plans on user request or output conversion.

3.2 Compiler and Runtime Operations

Compilation Chain (Fig. 3-2)**:** SystemDS inherits SystemML's compilation chain [7]. Each user script or DML-bodied function is compiled into a hierarchy of statement blocks and statements, where control flow statements like loops or branches delineate these blocks. All statements of a basic (i.e., last-level) block are compiled into a DAG of high-level operators (HOPs), which represent logical operations. After multiple rounds of rewrites, size propagation (of dimension and sparsity), and operator ordering, we then compute memory estimates for each operation. Based on these estimates, we in turn decide for local or distributed operations, and construct a DAG of low-level—i.e., physical—operators (LOPs). Finally, we create an executable program of program blocks and a sequence of runtime instructions—similar to MAL plans in MonetDB [36]—per block.

Runtime Control Program (Fig. 3-3)**:** The compiled runtime program is interpreted as a so-called control program (CP) in the client or Spark driver process. Besides program block and instruction execution, the control program
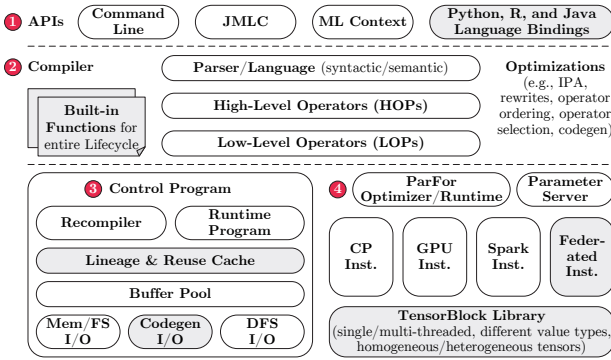
Figure 3: SystemDS Architecture and Components.

Figure 4: Example Tensor Representations.

also performs dynamic recompilation (recompilation of basic blocks to mitigate initial unknowns similar to adaptive query processing [19]), and maintains a multi-level buffer pool that is responsible for evicting intermediate variables if necessary, persistent reads and writes from and to distributed file systems like HDFS or S3, and data exchange between the different runtime backends. Major CP extensions are built-in support for data provenance and generated I/O primitives for external formats as discussed in Sections 4.1 and 4.2.

Runtime Operations (Fig. 3-4)**:** With the diversity of ML algorithms and apps (*L3*) in mind, we further extend SystemML's multiple backends. We include local CPU and GPU instructions, as well as distributed Spark instructions. In addition, we introduce a new class of federated instructions as discussed in Section 4.3. These instructions rely on a common `TensorBlock` operation library, which extends SystemDS from numeric matrices to heterogeneous, multi-dimensional arrays as described in Section 3.3. For local operations, such a block holds the entire tensor, while distributed tensors are represented as RDD collections of fixed-sized blocks. Besides reuse, this approach also ensures consistency across local and distributed operations. Additionally, we support dedicated backends for parallel for loops [6] (e.g., for hyper-parameter tuning, and cross validation), and parameter servers (e.g., for mini-batch DNN training). The changed data representation necessitates major changes throughout the entire compiler and runtime stack.

3.3 Data Model: Heterogeneous Tensors

Data Model Motivation: Our goal of supporting the end-to-end data science lifecycle poses two main requirements on the underlying data model. First, we need to represent heterogeneous and structured datasets for data integration and preparation (*L4*). Second, many lifecycle tasks and ML algorithms benefit from native support of multi-dimensional arrays. Therefore, and in contrast to most existing DNN frameworks and array databases—which support homogeneous arrays (e.g., tensors of floats or integers)—our data model is a heterogeneous tensor, that is, a multi-dimensional array where one dimension has a schema. We believe this is more natural than 2D datasets because it allows for range indexing that guarantees matching dimensions for subsequent operations (e.g., matrix multiplication).

Local Tensor: Our central data structure abstraction is a `TensorBlock` that represents a local tensor or a tile of a distributed tensor. Here, all single- and multi-threaded operations are implemented in Java for portability, but for compute-intensive operations we also support JNI calls to
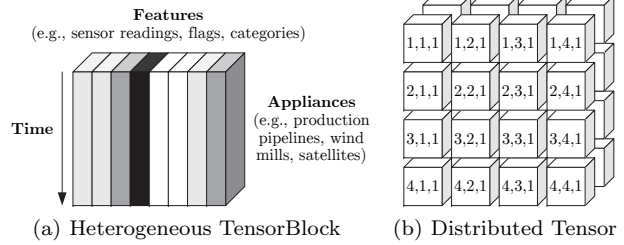
native BLAS libraries or custom C++ kernels. We provide two implementations of this `TensorBlock` abstraction:

- `BasicTensorBlock` *(Homogeneous):* A basic tensor is a linearized, multi-dimensional array of a single type (FP32, FP64, INT32, INT64, Bool, or String including JSON). We provide dense and sparse blocks and operations, which we apply based on the present sparsity.

- `DataTensorBlock` *(Heterogeneous):* A data tensor has a schema on the second dimension (see Figure 4(a)), which generalizes 2D datasets. Internally, it is composed of multiple basic tensors for the given schema.

Distributed Tensors: Our distributed tensor representation is a Spark RDD [78]—i.e., a distributed collection—of tensor indexes and fixed-size, independently-encoded blocks (`PairRDD<TensorIndexes,TensorBlock>`). Squared $1K \times 1K$ blocks in SystemML offer a good balance between amortized block overheads and moderate block sizes ($8\,MB$ for dense), simplify join processing because blocks are always aligned, and allow local transformations for operations like transpose. However, fixed-size blocking for n-dimensional data—as shown in Figure 4(b)—is challenging. We use a scheme of exponentially decreasing block sizes (1024^2, 128^3, 32^4, 16^5, 8^6, 8^7), which similarly bounds the size to few megabytes and allows for local conversion. For example, on a 3D-tensor/matrix operation, we split each 1024^2 matrix block into 64×128^2 blocks and perform the join, yielding again a 3D-tensor with 128^3 blocking.

Federated Tensors: For federated operations we provide a federated tensor that is a metadata object holding references to—potentially remote—in-memory or distributed tensors. Subtensors cover disjoint index ranges of the tensor (most commonly subsets of rows or columns), and uncovered areas are zero. This representation is the basis for federated learning, as discussed in Section 4.3, but also nicely generalizes operator placement to operations over multiple devices (e.g., 30% of data on CPU, 70% of data on GPU [28]).

4. KEY FEATURES AND DIRECTIONS

SystemDS shares several design aspects with other systems. In this section, we discuss some distinguishing features and research directions. However, we believe that building the overall system is of utmost importance for real impact and investigating these features in a realistic environment.

4.1 Lineage and Reuse of Intermediates

Efficient Lineage Tracing: Exploratory data science has a high degree of redundancy and most frameworks lack model versioning and reproducibility. Hence, we provide built-in support for data provenance in terms of lineage tracing and exploitation. We see lineage as a key enabling

technique for model versioning, reuse of intermediates, incremental maintenance, auto differentiation, more efficient buffer pool management, and debugging via query processing over lineage traces of different models or runs. In contrast to coarse-grained or data-oriented provenance, we focus on fine-grained lineage tracing of logical operations. We trace inputs (by name), literals, and all executed operations (including non-determinism like system-generated seeds) to maintain lineage DAGs of live variables. Additionally, for loops with few distinct control flow paths, we determine the lineage trace per path once, and track the taken path via a single lineage node for deduplication.

Reuse of Intermediates: Inspired by work on recycling intermediates in MonetDB [36], we then exploit this lineage for reusing redundantly computed intermediates, which are common in model selection workloads. We establish a cache, where intermediates are identified by their lineage (hash of the lineage DAG). Before executing an instruction, we update the output lineage and probe the cache for full or partial reuse. In contrast to existing work on coarse-grained reuse [23, 44, 68, 73, 77, 80], partial reuse computes an output via a compensation plan over cached intermediates. For example, `steplm` in Example 1 greedily adds features and performs what-if model training, which allows reusing intermediates from previous iterations, augmented by missing features.

Status: So far, we have integrated lineage tracing for local operations, lineage deduplication on while/for/parfor loops, basic caching and eviction policies, as well as full and partial reuse of intermediates. In the future, we intent to add rewrites for compiler-assisted reuse, more elaborate partial reuse, and query processing over collected lineage traces.

4.2 Data Integration and Cleaning

Semi-automated Data Preparation: Fully-automated data integration, cleaning and preparation is rather unrealistic given its complexity. We aim to provide abstractions (e.g., data extraction, semantic type inference, schema alignment, entity linking, outlier and anomaly detection, missing value imputation, data augmentation, and feature transformations) that help a user compose data preparation pipelines. Providing support for efficient and accurate data preparation faces, however, many algorithmic and systems challenges. We start by adding respective built-in functions, where we aim for vectorized implementations to simplify inference and optimization as well as search space pruning via sparsity exploitation. For example, masking allows data slicing and missing value imputation (in chained-equation models [71]) via sequences of full matrix operations, which significantly simplifies the compilation into multi-threaded or distributed runtime plans. Overall, a key design choice is to retain the appearance of a stateless system by consuming pre-trained models and rules as tensors themselves.

Efficient Data Ingestion: Given these abstractions, efficient ingestion faces two more challenges. First, the number of external data formats is virtually unlimited and sometimes requires even custom parsers for nested data. Inspired by work on query processing over CSV, JSON, and binary data [38], we aim to automatically generate code for efficient readers and writers from high-level descriptions of data formats. In this context, we further aim to avoid unnecessary parsing [53], and unnecessary shuffling [42] by taking the entire preparation pipeline into account. Second, semi-automated data preparation is still an exploratory process.

Similar, to query processing over raw data [2], we aim to exploit the lineage-based reuse of intermediates and build dedicated access methods for linear algebra over raw data in multi-tenant data science workflows and federated ML.

Status: We already added built-in functions for schema detection, outlier detection, missing value imputation, data augmentation, model debugging, as well as additional input formats and feature transforms. In the future, we will incorporate state-of-the-art algorithms, improve accuracy for structured datasets, and focus on efficient data ingestion.

4.3 Federated ML

Motivation: Early work on federated learning [10, 47] shows great promise, but focuses on mini-batch ML algorithms over private data from mobile devices. We believe federated ML is broadly applicable in the enterprise as well, a view shared by recent work on enterprise model fusion [75]. First, it could create a spectrum of data ownership and sharing (private data, shared gradients/aggregates, shared data) enabling new markets and business models. Second, it could enable ML in geo-distributed or restricted environments, where data consolidation is infeasible.

Federated ML Architecture: Our basic design consists of multiple control programs, each having local data. A master control program holds the federated tensors (see Section 3.3) including connections to the remote workers waiting for commands. SystemDS then allows both, cross-datacenter federation [76] (where each control program runs in a Spark cluster) as well as federation of individual endpoints. We aim to support linear algebra operations (and thus, all abstractions from Figure 1), as well as distributed parameter servers over federated tensors. Special federated instructions process these federated tensors by pushing as much computation to the remote sites as possible, while complying with exchange constraints and leveraging means of cryptography[2]. We will further extend our existing parameter server to respect the boundaries of federated tensors as well.

EXAMPLE 2 (FEDERATED MV MULTIPLICATION). *For example, consider matrix-vector (MV) and vector-matrix (VM) multiplications on a federated matrix, where each site holds a partition of rows. For an MV multiplication, the master broadcasts the vector to the workers, lets them compute a local MV, collects the result vectors, and constructs the output vector via* `rbind`. *For a VM multiplication, the master sends only relevant vector slices to the workers, lets them compute a local VM, collects the result vectors, and computes the output vector by adding the individual results.*

Status: So far, we have a basic integration of federated tensors and selected federated operations. In the future, we will focus on broad operation coverage, data preparation and cleaning of raw input data, efficient data exchange and materialization, as well as exchange constraints.

4.4 Compiler and Runtime Improvements

The stack of declarative abstractions from Figure 1 requires major extensions of the compiler and runtime. We are interested in the following related research directions:

- *ML & Rules:* Complex ML apps often combine ML models and rules in meta models, which require dedicated compilation and verification techniques.

---

[2]For example, homomorphic encryption [26] allows multiply and add (and thus, matrix multiply) directly over encrypted data.

- *Size Propagation:* Propagating dimensions [7] and sparsity [67]—or at least lower and upper bounds—through control flow of the entire lifecycle is challenging but essential for cost-based optimization.
- *Operator Fusion & Code Generation:* Fusion is a widely recognized optimization, but the potential for sparsity exploitation [9] is barely leveraged yet.
- *Lossless and Lossy Compression:* Recent work on lossless compression for linear algebra [21, 45] and quantization for DNN workloads need a systematic investigation regarding data tensors and federated ML.
- *Cloud and Auto Scaling:* The stateless design and size inference also enable automatic resource optimization [34] in cloud environments, which is still an obstacle.

5. PRELIMINARY EXPERIMENTS

Our experiments study the baseline performance of SystemDS and optimization opportunities across lifecycle tasks.

5.1 Experimental Setting

We ran all experiments on a single node with two Intel Xeon E5-2620 CPUs @ 2.10-2.50 GHz (24 virtual cores), 128 GB DDR3 RAM, and CentOS Linux 7.4. SystemDS 0.1+ (as of 12/2019, v0.1 released 08/2019, forked from SystemML 1.2 in 09/2018) uses OpenJDK 1.8.0 with 80 GB max and initial JVM heap sizes. The baselines are TensorFlow 1.13.2 (07/2019), TensorFlow 2.0.0 (10/2019), and Julia 1.1.1 (05/2019). The workloads are (1) a hyper-parameter optimization script (HPO) that reads a CSV file, trains k regression models with different regularization λ (see `lmDS` in Figure 2), and stores the resulting models as a CSV file, and (2) a cross-validation script (CV) that reads a CSV file, runs k-fold cross-validation for `lmDS`, and stores all k models as a CSV file. We generate synthetic dense and sparse data, use optimized TF and Julia scripts, and report the end-to-end runtime including I/O as the mean of 3 repetitions.

5.2 Baseline Comparison

For evaluating the baseline performance of SystemDS, we use a 100K × 1K matrix $\mathbf{X}$ (800 MB in-memory, 1.79 GB CSV file) and train $k \in (1, 10, 20, 30, 40, 50, 60, 70)$ models. The main computation of `lmDS` is $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$, which requires 100.2 GFLOP per model but is independent of λ. Figures 5(a) and 5(b) compare TensorFlow 1.3 with NumPy array (TF) and tensor (TF-G) outputs—where the latter constructs a single graph and thus, can eliminate common subexpressions—Julia, SystemDS (SysDS) and SystemDS with native Intel MKL BLAS library (SysDS-B). There are four main observations. First, multi-threaded I/O in SysDS yields better performance than TF or Julia for a single model because string-to-double parsing is compute-intensive. Second, our multi-threaded, cache-conscious Java matrix multiplications show good performance but are 2.1x slower than Julia's native operations because Java does not compile packed SIMD instructions. With native BLAS for dense matrix-matrix multiplication, SysDS-B then slightly outperforms Julia. For TF, we had to manually rewrite `tf.matmul(tf.matrix_transpose(X), X)` into a fused API call to avoid excessive transpose costs. Third, Figure 5(b) shows that SysDS largely outperforms Julia and TF on sparse data (with sparsity=nnz/#cells=0.1). TF has large transpose overhead as its sparse-dense matrix multiply lacks

(a) Baselines Dense (b) Baselines Sparse
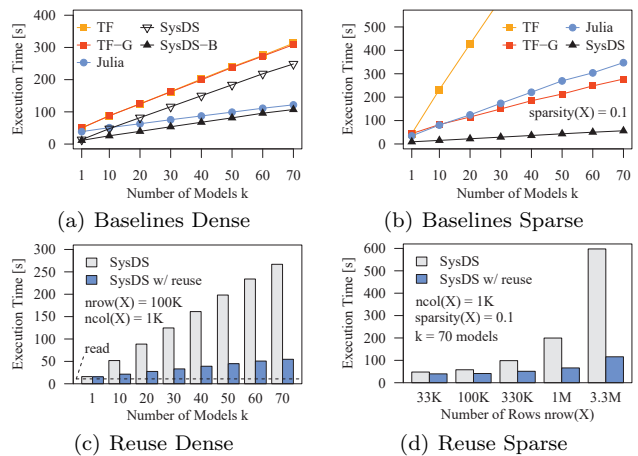(c) Reuse Dense (d) Reuse Sparse

Figure 5: SystemDS Baseline Comparisons.

a fused call, while TF-G executes this transpose only once. Fourth, and most importantly, none of these systems is able to eliminate the redundant matrix multiplications.

5.3 Reuse of Intermediates

Figure 5(c) shows the SystemDS results—for HPO over a 100K × 1K dense input matrix $\mathbf{X}$—with enabled lineage-based reuse as described in Section 4.1. For one model, there is no redundancy. As the number of models increases, however, we see substantial improvements by reusing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Despite the I/O of 10.9s and several operations that are not subject to reuse, we get a 4.6x end-to-end speedup for 70 models. Figure 5(d) shows the impact of input data sizes by varying the number of rows in $\mathbf{X}$ (with sparsity=0.1). The larger the input, the higher the improvements because the remaining operations access only intermediates, whose size is independent of the number of rows.

5.4 TensorFlow 2 Comparison

HPO: We further compare SystemDS with the recently released TensorFlow 2.0 (compiled from sources), which introduced eager execution (TF) and lazy evaluation via AutoGraph [49] (TF-G). Due to dependency conflicts, we ran these experiments in a VM with 80 GB RAM, 4 virtual cores, Ubuntu 18.04, and 70 GB max and initial JVM heap sizes. Figure 6 shows the results for HPO, where TF-G is now able to reuse intermediates as well, while still causing substantial overhead for sparse data. Both TF and TF-G ran into segmentation faults for Figure 6(b), 3.3M though.

CV: Figure 7 compares SystemDS 0.1+ and TensorFlow 2.0 on the CV use case (k-fold cross validation, with leave-one-out). On dense data, TF/TF-G perform slightly better than SysDS-B, while on sparse data, TF/TF-G show again substantial overhead (log scale). Most importantly,
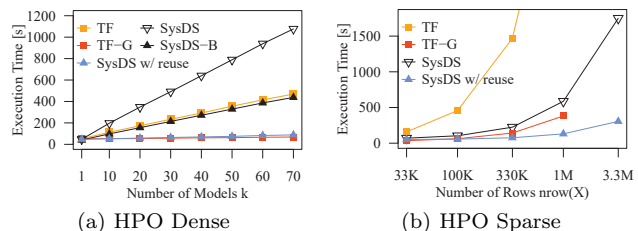

(a) HPO Dense (b) HPO Sparse

Figure 6: SystemDS-TF2 Comparison (HPO).
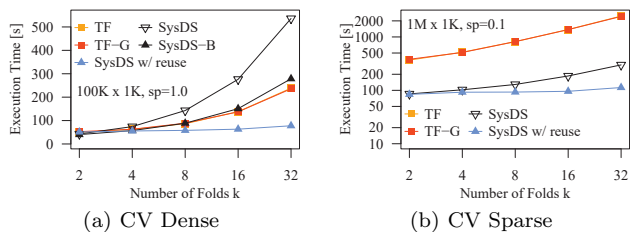
(a) CV Dense   (b) CV Sparse

Figure 7: SystemDS-TF2 Comparison (CV).

only SysDS with reuse eliminates the fine-grained redundancy, which would be difficult in the AutoGraph model. Full reuse relies on rewriting `X=rbind(remove(foldsX,i))`, `y=rbind(remove(foldsY,i))`, and the matrix multiplications $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ during dynamic recompilation into multiplications of the individual folds (which are subject to reuse) and element-wise addition of these intermediates.

6. RELATED WORK

SystemDS has a broad focus and thus, there is lots of related work for individual aspects. Therefore, we focus on systems for data science lifecycle tasks and array processing.

ML Systems for Data Science: Several recent systems also aim to support the data science lifecycle in a scalable manner. First, Northstar [40]—a collection of tools for interactive data science—includes Alpine Meadow [65] for automatic feature pre-processing and AutoML based on existing ML libraries (e.g., Scikit-learn) and custom operators. Second, TensorFlow Extended (TFX) [3] provides components for data ingestion, validation, transformation, as well as model training, validation and serving. These components have different backends (e.g., transform in Apache Beam, train in TensorFlow) and are composed via orchestration tools like Apache Airflow or Kubeflow. Third, MLflow [79] provides means of model management (e.g., tracking experiments), project packaging, and model deployment. Alpine Meadow, TFX, and MLflow rely on existing ML libraries, while Weld [52] focuses on a minimalistic but invasive, UDF-based IR for optimizing across different libraries. In contrast, SystemDS builds on a common DSL, provides its own compiler and runtime and thus, can exploit fine-grained optimization opportunities. Fourth, systems like AIDA [66] and Lara [43] aim at joint relational and linear algebra in data-science-centric specification languages that are mapped to a common IR and then executed via existing SQL engines, data-parallel frameworks, or numerical computing libraries. The design of SystemDS differentiates by support for tensors, distributed and federated linear algebra, and broad support for the end-to-end data science lifecyle.

Array Processing: Array databases (e.g., SciDB [69]) and array libraries (e.g., NumPy [72], DL4J/NDArray) focus primarily on scientific computing and respective formats. While array databases require loading and schema design for efficient distributed operations, data scientist seem to favor stateless systems and functional R or Python libraries and DSLs. Scalability limitations are addressed by new Python libraries like dask [18] and xarray [33] for distributed, multi-dimensional array processing. Although these libraries do not optimize for ML workloads, they are increasingly used by scikit-learn to provide distributed ML algorithms. In contrast, SystemDS aims to provide abstractions for a variety of data science lifecycle tasks and users, as well as efficient linear algebra and optimization across the lifecycle.

7. CONCLUSIONS

To summarize, we described the vision and system architecture of SystemDS, an open-source ML system for end-to-end data science pipelines. Compared to SystemML, the major differences are (1) support for data science lifecycle tasks (e.g., data preparation, training, and debugging) and users with different expertise (ML researchers, data scientist, and domain experts), (2) support for local, distributed, and federated ML, and (3) the data model of heterogeneous data tensors. We also outlined selected research directions, and promising preliminary results. SystemDS is early work-in-progress, but throughout the next years (or decades), we will continuously improve it, leverage it in real-world applications, and use it for grounding our research in a real system. We encourage the DB and ML systems community to use SystemDS as a baseline or testbed for extensions.

8. REFERENCES

[1] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[3] D. Baylor et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *SIGKDD*, 2017.

[4] P. A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *SIGMOD*, 2007.

[5] J. Bezanson et al. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1), 2017.

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

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

[8] M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.

[9] M. Boehm et al. On Optimizing Operator Fusion Plans for Large-Scale ML in SystemML. *PVLDB*, 11(12), 2018.

[10] K. Bonawitz et al. Towards Federated Learning at Scale: System Design. In *SysML*, 2019.

[11] J. Cambronero et al. Query Optimization for Dynamic Imputation. *PVLDB*, 10(11), 2017.

[12] V. Chandola et al. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41(3), 2009.

[13] T. Chen et al. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[14] Y. Cheng et al. GLADE: Big Data Analytics Made Easy. In *SIGMOD*, 2012.

[15] Y. Chung et al. Slice Finder: Automated Data Slicing for Model Validation. In *ICDE*, 2019.

[16] J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[17] E. D. Cubuk et al. AutoAugment: Learning Augmentation Policies from Data. In *CVPR*, 2019.

[18] Dask. *Dask: Library for dynamic task scheduling*, 2016.

[19] A. Deshpande et al. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1), 2007.

[20] X. L. Dong and T. Rekatsinas. Data Integration and Machine Learning: A Natural Synergy. In *SIGMOD*, 2018.

[21] A. Elgohary et al. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12), 2016.

[22] X. Feng et al. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, 2012.

[23] R. C. Fernandez et al. Meta-Dataflows: Efficient Exploratory Dataflow Jobs. In *SIGMOD*, 2018.

[24] M. Feurer et al. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *AML*. 2019.

[25] K. E. Gebaly et al. Interpretable and Informative Explanations of Outcomes. *PVLDB*, 8(1), 2014.

[26] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

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

[28] M. Gowanlock et al. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *DaMoN*, 2019.

[29] J. Heer et al. Predictive Interaction for Data Transformation. In *CIDR*, 2015.

[30] A. Heidari et al. HoloDetect: Few-Shot Learning for Error Detection. In *SIGMOD*, 2019.

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

[32] V. J. Hodge and J. Austin. A Survey of Outlier Detection Methodologies. *Artif. Intell. Rev.*, 22(2), 2004.

[33] S. Hoyer and J. Hamman. xarray: N-D labeled Arrays and Datasets in Python. *JORS*, 5(1), 2017.

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

[35] M. Hulsebos et al. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. In *KDD*, 2019.

[36] M. Ivanova et al. An Architecture for Recycling Intermediates in a Column-store. In *SIGMOD*, 2009.

[37] S. Kandel et al. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*, 2011.

[38] M. Karpathiotakis et al. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12), 2016.

[39] L. Kotthoff et al. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *JMLR*, 18, 2017.

[40] T. Kraska. Northstar: An Interactive Data Science System. *PVLDB*, 11(12), 2018.

[41] A. Kumar et al. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*, 2015.

[42] A. Kunft et al. BlockJoin: Efficient Matrix Partitioning Through Joins. *PVLDB*, 10(13), 2017.

[43] A. Kunft et al. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB*, 12(11), 2019.

[44] Y. Lee et al. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *OSDI*, 2018.

[45] F. Li et al. Tuple-oriented Compression for Large-scale Mini-batch SGD. In *SIGMOD*, 2019.

[46] S. Luo et al. Scalable Linear Algebra on a Relational Database System. In *ICDE*, 2017.

[47] B. McMahan et al. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*, 2017.

[48] X. Meng et al. MLlib: Machine Learning in Apache Spark. *JMLR*, 17, 2016.

[49] D. Moldovan et al. AutoGraph: Imperative-style Coding with Graph-based Performance. In *SysML*, 2019.

[50] M. Nikolic and D. Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, 2018.

[51] R. S. Olson and J. H. Moore. TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning. In *AML*. 2019.

[52] S. Palkar et al. A Common Runtime for High Performance Data Analysis. In *CIDR*, 2017.

[53] S. Palkar et al. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *PVLDB*, 11(11), 2018.

[54] L. Passing et al. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*, 2017.

[55] A. Paszke et al. Automatic differentiation in PyTorch. 2017.

[56] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

[57] F. Pedregosa et al. Scikit-learn: Machine Learning in Python. *JMLR*, 12, 2011.

[58] N. Polyzotis et al. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record*, 47(2), 2018.

[59] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*, 2001.

[60] T. Rekatsinas et al. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB*, 10(11), 2017.

[61] S. Schelter et al. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. *NIPS MLSys*, 2016.

[62] S. Schelter et al. Automating Large-Scale Data Quality Verification. *PVLDB*, 11(12), 2018.

[63] M. Schleich et al. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, 2016.

[64] D. Sculley et al. Hidden Technical Debt in Machine Learning Systems. In *NIPS*, 2015.

[65] Z. Shang et al. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*, 2019.

[66] J. V. D. silva et al. AIDA - Abstraction for Advanced In-Database Analytics. *PVLDB*, 11(11), 2018.

[67] J. Sommer et al. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*, 2019.

[68] E. R. Sparks et al. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*, 2017.

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

[70] M. Stonebraker and I. F. Ilyas. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.*, 41(2), 2018.

[71] S. van Buuren and K. Groothuis-Oudshoorn. mice: Multivariate Imputation by Chained Equations in R. *J. of Stat. Software 2011*, 20(1), 2011.

[72] S. van der Walt et al. The NumPy Array: A Structure for Efficient Numerical Computation. *Comp.S&E*, 13(2), 2011.

[73] M. Vartak et al. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*, 2018.

[74] W. N. Venables and B. D. Ripley. *Modern applied statistics with S, 4th Ed.* Statistics and computing. Springer, 2002.

[75] D. C. Verma et al. Federated AI for the Enterprise: A Web Services Based Implementation. In *ICWS*, 2019.

[76] A. Vulimiri et al. WANalytics: Analytics for a Geo-Distributed Data-Intensive World. In *CIDR*, 2015.

[77] D. Xin et al. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB*, 12(4), 2018.

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

[79] M. Zaharia et al. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.*, 41(4), 2018.

[80] C. Zhang et al. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.

[81] D. Zhang et al. Sato: Contextual Semantic Type Detection in Tables. *CoRR*, abs/1911.06311, 2019.