A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics

Anthony Thomas   Arun Kumar

University of California, San Diego

{ahthomas, arunkk}@eng.ucsd.edu

ABSTRACT

The growing use of statistical and machine learning (ML) algorithms to analyze large datasets has given rise to new systems to scale such algorithms. But implementing new scalable algorithms in low-level languages is a painful process, especially for enterprise and scientific users. To mitigate this issue, a new breed of systems expose high-level bulk *linear algebra* (LA) primitives that are scalable. By composing such LA primitives, users can write analysis algorithms in a higher-level language, while the system handles scalability issues. But there is little work on a unified comparative evaluation of the scalability, efficiency, and effectiveness of such "scalable LA systems." We take a major step towards filling this gap. We introduce a suite of LA-specific tests based on our analysis of the data access and communication patterns of LA workloads and their use cases. Using our tests, we perform a comprehensive empirical comparison of a few popular scalable LA systems: MADlib, MLlib, SystemML, ScaLAPACK, SciDB, and TensorFlow using both synthetic data and a large real-world dataset. Our study has revealed several scalability bottlenecks, unusual performance trends, and even bugs in some systems. Our findings have already led to improvements in SystemML, with other systems' developers also expressing interest. All of our code and data scripts are available for download at `https://adalabucsd.github.io/slab.html`.

1. INTRODUCTION

Supporting large-scale statistical and machine learning (ML) algorithms over structured data has become a mainstream requirement of modern data systems [1, 8]. Consequently, both the database and cloud service industries are scurrying to make it easier to integrate ML algorithms and

frameworks such as R with data platforms [2, 9, 13, 14, 16, 25, 41]. While a roster of canned implementations might suffice for popular existing algorithms, many application users, especially in the enterprise and scientific domains often create custom analysis algorithms for their data. Such users typically write their algorithms using *linear algebra* (LA) notation and are accustomed to tools such as R, SAS, and Matlab [11, 17]. LA is a succinct and elegant mathematical language to express matrix transformations for a wide variety of statistical and ML algorithms [25, 30]. However, such LA tools are mostly not scalable to datasets that may not fit in single-node memory, which impedes effective analytics as the volume of datasets continues to grow.

The above situation has led to a new breed of analytics systems: *scalable* LA systems, which allow users to scale their LA-based algorithms to distributed memory-based or disk-based data without needing to manually handle data distribution and communication. Thus, they offer a measure of *physical data independence* for LA-based analytics. Examples include RIOT [71], EMC's MADlib [41], Oracle R Enterprise [16], Apache SystemML [25], Microsoft's Revolution R [14], Spark MLlib [55] and SparkR [18], Apache Mahout Samsara [62], LA on SimSQL [51], and Google's TensorFlow [20]. Most of these systems expose an LA-based language or API embedded in Python, Scala, or SQL as their front-end. Almost all of them aim to scale LA operations that process the whole dataset *in bulk*, i.e., in one go in a data-parallel manner. A common example of an algorithm that can be expressed in bulk LA is least-squares linear regression (Algorithm 1). This access pattern differs from that of stochastic gradient descent-based training of neural networks, which is not a bulk LA operation but rather repeatedly samples data subsets called mini-batches [56]. Thus, except for TensorFlow, the above systems do not (yet) offer strong support for neural networks, although there are active efforts to extend some of them [60]. Accordingly, the focus of this paper is on systems for bulk LA workloads.

While the recent activity on scalable LA systems has led to many interesting research papers and open-source systems, a fundamental practical question remains largely unanswered: *From a comparative standpoint, how effective and efficient are such systems for scalable LA and LA-based ML?* By effectiveness, we mean how "easy" they are to use in terms of the amount of code needed to express algorithms using their LA abstractions, as well as their setup logistics. By efficiency, we mean how "fast" they run in different settings, especially the increasingly popular distributed memory setting. Different systems have different assumptions and re-

strictions in their programming and execution models, leading to different runtime and memory usage behaviors. Existing papers and articles on such systems mostly focus on the operating regimes where they perform well, leaving users in the dark on where they may not perform so well. Additionally, most systems were not compared to strong baselines such as pbdR (powered by ScaLAPACK) [59]. This *lack of a unified comparative understanding* of such systems' behaviors could lead to both user frustration and increased costs when picking a system to use, especially in the cloud.

We take a step towards filling this gap by introducing a suite of performance tests for scalable LA systems and performing a comprehensive empirical comparison of several popular and state-of-the-art LA systems using our tests. Our test suite is inspired by the long line of work on benchmarks for LA packages in both the single-node in-memory and distributed settings, e.g. benchmarks comparing BLAS, Eigen, Intel MKL, LAPACK, and ScaLAPACK [21, 23, 34, 35, 48, 57]. But unlike such prior work, which focused only on simple LA operations, we want to evaluate the entire hierarchy of task complexity for LA-based analytics. Thus, we delineate three orthogonal axes of interest: *task complexity*, *data scale*, and *computation environment*. For data scale, we vary both the number of examples and sparsity of the data matrix. For the computation environment, we focus on commodity CPU clusters (as of this writing, few of the compared systems offer easy integration with GPUs, FPGAs, or other hardware accelerators in the distributed setting). We vary the number of CPU cores and cluster nodes.

For task complexity, we include 6 common LA operations with differing communication and computation behaviors, as well as 2 simple pipelines of LA operations and 4 LA-based data analysis algorithms, including 2 popular ML algorithms. The tasks at the higher levels of complexity present interesting opportunities for *inter-operation optimizations* (akin to relational query optimization), which are exploited by only some of the scalable LA systems. Moreover, the LA-based algorithms enable us to qualitatively compare how easy or difficult it is to use the LA abstractions provided by such systems to write such algorithms. The LA-based algorithms we include also cover a diverse set of use cases for scalable LA-based statistical and ML analytics: regression, heteroscedasticity-aware error analysis, classification, and feature extraction.

We compare the following systems: MADlib, MLlib, SystemML, ScaLAPACK, SciDB, TensorFlow, R, and NumPy. We picked them mainly due to their popularity but also because they are open source and have active user communities online. Our experimental study has revealed numerous interesting behaviors at large data scales, including unexpected tuning headaches, hidden scalability bottlenecks, unusual relative performance trends, and even bugs in some systems. For instance, we find that tuning memory-related parameters correctly is crucial to get good performance with the Spark-based systems (MLlib and SystemML), but the optimal settings are LA task-specific, making it a non-trivial task for data scientists. RDBMS-based MADlib yielded relatively poor performance for LA operations on dense data and suffers from a scalability bottleneck due to arcane restrictions imposed by the RDBMS, but it is competitive on sparse data. Even more surprisingly, pbdR/ScaLAPACK outperforms all other systems in almost all cases on dense data. Overall, we find that SystemML offers perhaps the best balance between physical data independence and performance at scale among the compared systems.

Based on our empirical study, we identify and summarize the strengths and weaknesses of each scalable LA system to help practitioners decide which one best fits their application needs. We also identify the major gaps in the capabilities of such systems and distill them into a handful of open research questions for those interested in improving such systems or building new ones. In particular, we find that support for large sparse datasets is still relatively poor; this is a pressing issue due to the ubiquity of large-domain categorical features in ML applications. We also find that many systems sometimes exhibit sub-linear multi-node speedups. We suggest that flexible hybrid parallelism models combining data and task parallelism could make such systems more useful for crucial meta-level ML tasks such as hyper-parameter tuning [45]. Overall, this paper makes the following contributions.

- To the best of our knowledge, this is the first work to create a unified framework for comparative evaluation of popular scalable LA systems. We discuss key design considerations in terms of data access, communication and computation patterns, and use cases, and we create a suite of tests spanning the axes of task complexity, data scale, and computational environment.
- Using our tests, we perform an extensive empirical study comparing MADlib, MLlib, SystemML, pbdR/ScaLAPACK, and SciDB in the distributed memory setting, along with TensorFlow, R, and NumPy on a single node. Apart from synthetic data of various scales, we create two benchmark versions of a large real-world dataset from an online advertising firm, Criteo [5].
- We analyze and distill our results into a concrete set of guidelines for practitioners interested in using such systems. We also identify a handful of major open questions for further research on such systems.
- Our findings have already resulted in bug fixes and feature earmarks for SystemML due to our conversations with its developers. We are also speaking with the developers of some of the other systems to help them improve their systems using our results.

To improve reproducibility and to help others extend our work, all of our code for all tests on all systems compared, including configuration files, as well as all of our scripts for generating synthetic data and pre-processing Criteo data are available for download on our project webpage: `https://adalabucsd.github.io/slab.html`.

Outline. Section 2 provides some background on LA systems and overviews of the systems we compare. Section 3 explains our test suite. Section 4 presents our empirical study with both quantitative results and qualitative discussion. Section 5 presents more analysis and discussion of the implications of our findings for both practitioners and researchers. We discuss other related work in Section 6.

2. BACKGROUND

2.1 LA Systems

A data matrix $\mathbf{X}_{n \times d}$ has n examples and d features. Linear algebra (LA) is elegant and expressive formal language

to capture linear transformations of $\mathbf{X}$ (vectors and scalars are special cases). Numerous statistical and ML algorithms can be expressed using LA operations that are "bulk" vectorized computations over $\mathbf{X}$, including such popular algorithms as ordinary least squares (OLS) linear regression, logistic regression solved with many gradient methods [58], non-negative matrix factorization, k-means clustering, and more [30, 37, 39]. Such algorithms are common in enterprise analytics, as well as social and physical sciences.

Scalable LA systems aim to let users write LA-based algorithms but scale them to larger-than-memory datasets *transparently*, i.e., with some degree of *physical data independence*. Essentially, such systems abstract away lower-level systems issues such as parallel computation, data communication, and fault tolerance (in some cases). In a sense, scalable LA systems aim to achieve for LA what RDBMSs achieve for relational algebra. Different systems take different routes: some layer LA as an abstraction on top of a RDBMS or Spark, while others implement LA primitives in standalone systems. Our goal is *not* to design new scalable LA systems, but to provide a systematic quantitative and qualitative comparison of state-of-the-art and popular systems on an even footing, using a carefully designed suite of tests at multiple levels of abstraction.

2.2 Overview of Compared Systems

Rationale for Systems Picked. We restrict our study to systems that are open source, well-documented, and have helpful online user communities. This enables us to properly debug unusual system behaviors. Thus, we skip other major open-source systems (e.g., RIOT, Samsara, SimSQL, and RASDAMAN). We also skip industrial tools such as Oracle R Enterprise and Microsoft Revolution R for legal reasons but hope that our work helps spur more industrial standardization in this arena. We study two popular system environments: single-node in-memory and distributed memory (with disk spills allowed) but focus on the latter. For the former, R and Python's NumPy are the most popular tools [11], with TensorFlow being a new contender. For the latter, the primary scalable LA systems are pbdR, powered by ScaLAPACK [23, 59], the array database SciDB [27], the distributed RDBMS-based MADlib [41], and Spark-based MLlib (or SparkML) [55] and SystemML [25]. TensorFlow does not yet offer distributed implementations of bulk LA operations [7]; thus, we skip it for the distributed setting.

R and NumPy. These libraries treat matrices as first-class citizens, offer a rich set of built-in LA operations and algorithms, and are widely used in academia and enterprises [11]. Most LA operations in R and NumPy are thin wrappers around highly optimized LAPACK and BLAS routines [21, 32]. R and Python also provide robust visualization/plotting libraries (e.g., ggplot and matplotlib). Both R and Python are interpreted and dynamically typed, which makes optimization of LA scripts challenging.

pbdR (ScaLAPACK). ScaLAPACK extends LAPACK to distributed memory by reimplementing many LA operations and algorithms using a block partitioning scheme for matrices [23]. It follows the "same program multiple data" paradigm in which one logical program is executed by multiple workers (a worker corresponds to a core). A matrix is partitioned in a highly flexible "block-cyclic" fashion (similar to round robin), with each worker process allocated a

subset of the blocks. This allocation helps load-balance computation and communication costs regardless of the access patterns of the LA operations. The block size is a user-given parameter. The "Programming with Big Data in R" (pbdR) library provides higher level R interfaces to ScaLA-PACK and OpenMPI [59]. The "distributed matrix" package in pbdR overloads several built-in LA operations in R to enable transparent distributed execution. pbdR compiles programs to MPI batch jobs, which are then submitted for execution on an MPI cluster [38].

MADlib. MADlib is a library that implements both LA primitives and popular ML algorithms over PostgreSQL and the parallel RDBMS Greenplum [41]. Matrices are stored as tables and many LA operations are implemented directly in SQL, enabling MADlib to exploit the RDBMS for memory management and scalability. Some operations, such as eigen or singular value decompositions, require the matrix to fit in single-node memory [52]. Related to MADlib (but not compared here) are RIOT-DB [71], which translates LA scripts written in R into SQL, and SimSQL [51] which implements custom datatypes for block-partitioned matrices and custom user-defined functions for LA operations.

Spark MLlib/SparkML. These are libraries that provide LA primitives and canned implementations of some popular ML algorithms on Spark RDDs. We use MLlib, since SparkML does not yet support distributed matrices. Apart from the *LocalMatrix* datatype for small data, MLlib has three main (physical) datatypes for distributed matrices targeting different data access patterns. The user must select a matrix type explicitly. The operators provided for each type are not consistent, e.g., the type supporting matrix-matrix multiplication does not support decompositions and vice versa. All matrix types support either sparse or dense data, but some operators densify sparse matrices.

SystemML on Spark. Introduced for Hadoop and then ported to Spark, SystemML is perhaps the most mature scalable LA system [25]; we use the recommended Spark version [4]. SystemML offers a "declarative" language called DML with R-like syntax to express LA scripts (there are also APIs in Python and Scala/Java). DML offers full physical data independence, i.e., users do not decide data layout formats or low-level execution details. SystemML stores matrices in a block-partitioned format implemented as a Spark RDD. Inspired by RDBMSs, SystemML has an optimizing compiler that converts a DML script to Spark jobs by applying a suite of logical LA-specific rewrite optimizations and physical execution optimizations. The compiler translates a DML script into a physical execution plan which selects operator implementations based on data and system characteristics (e.g. size, sparsity, RAM). SystemML can transparently switch between local and distributed execution.

TensorFlow. TensorFlow (TF) is a framework for expressing ML algorithms, especially neural networks [20]. It has APIs in Python and `C++` for both LA primitives and canned ML implementations; we use the Python API. While TF is primarily meant for easily expressing and training complex neural network architectures using mini-batch stochastic gradient decent (SGD), it can also be used for bulk LA-based algorithms in the single-node setting. But does not provide scalable LA operations. A TF program is modeled as a graph in which nodes represent operations over multi-

dimensional arrays ("tensors") and edges represent dataflow. TF allows for operators to execute on different cluster nodes and handles communication under the hood. Thus, TF is able to automatically scale mini-batch algorithms to the cluster setting, but it cannot scale bulk LA algorithms, since each *individual* operator must fit on a single node.

SciDB. SciDB is a database system aimed at scientific computing applications [27]. It provides a declarative interface similar to SQL in which users compose algebraic operators that can read, write, and update multidimensional arrays. Internally, SciDB shards a logical array into a set of possibly overlapping chunks that are distributed over instances that possibly reside on different physical nodes. As of this writing, SciDB supports a fairly limited set of LA operations (matrix-matrix multiplication and singular value decomposition). Furthermore, these operations do not support disk spills, i.e., the data must fit in distributed memory.

3. DESCRIPTION OF TESTS

We delineate our tests along three orthogonal axes; for each axis, we list parameters to vary in order to test both the computational and communication aspects of scalable LA workloads. We now explain each axis in detail.

Axis 1: Task Complexity. At a high-level, we decompose bulk LA workloads into three levels of task complexity. Each level of complexity targets different kinds of implementation choices and optimizations (if any) in scalable LA systems. The first, and lowest, level of tasks consists of basic LA operations over the data matrix. Such operators are the bedrock of LA-based ML; we denote this set of tests **MAT**. The second level consists of "simple" pipelines (compositions) of LA operations, which offer more scope for optimization; we denote this set by **PIPE**. The highest level of task complexity consists of LA-based statistical analysis and ML algorithm scripts; we denote this set by **ALG**. Section 3.1 explains the specific tasks we pick for each level in detail and why. Table 1 summarizes all tasks.

Axis 2: Data Scale Factors. Varying key properties of the data matrix lets us stress test how the systems perform at different data scales. For the sake of simplicity and uniform treatment across all systems, we assume the data matrix X has only of numeric features; categorical features are assumed to be pre-converted using one-hot encoding to obtain sparse 0/1 vectors [15]. There are three key settings: number of rows, number of columns and the fraction of non-zero cells, also called *sparsity*. We note that the number of rows and columns governs the shape of the data matrix X. We focus on varying the number of rows and data sparsity as these parameters govern most practical workloads.

Axis 3: Computational Scale Factors. This axis captures the amount of computational resources available. Once again, for the sake of simplicity and uniform treatment across all systems, we focus on commodity CPU-based compute clusters in the cloud. Thus, we only vary two settings: number of CPU cores and number of worker nodes. While this is a popular setting for many enterprises and domain scientists, we note that GPUs, FPGAs, and custom ASICs (e.g., TPUs [43]) can be used to accelerate LA workloads. However, not all scalable LA systems currently support such

hardware. Thus, we leave it to future work to compare the systems in these new environments.

3.1 Task Complexity

While data and computational scale factors are standard for any benchmark comparison of data systems, the task complexity axis is LA-specific. We delineate computational and communication behaviors of scalable LA workloads and capture them using a small set of tests at three levels of complexity, summarized by Table 1. We now elaborate upon our rationale for picking these specific tests.

3.1.1 MAT: *Matrix Operations*

Scalable implementations of LA operations are the bedrock of scalable LA systems. But there are far too many LA operations for an exhaustive list to be practical. We resolve this issue by observing that LA operations can be grouped into a small set of categories based on commonalities in their data access patterns, memory footprints, and computational and communication behaviors on large data matrices partitioned across workers in a pre-specified way (row-wise, column-wise, or block-partitioned). We define six major categories:

(1) Unary operations that only need partition-local local reads/writes without communication, e.g., matrix transpose and scalar-matrix multiplication. If the partitioning is row-wise (resp. column-wise), row-wise (resp. column-wise) summation also belongs to this category.

(2) Unary operations that require communicating a constant-sized state between workers, e.g., norms, trace, and full summation. These are analogous to algebraic aggregates in SQL.

(3) Unary operations that require communicating a state of data-dependent size between workers, typically as shuffles, e.g., Gramian and outer product. Partial summations that do not belong to the first category belong here.

(4) Binary operations in which one of the inputs typically fits in single-node memory, e.g., matrix-vector multiplication.

(5) Binary operations in which both are inputs larger than single-node memory but co-partitioning can avoid communication, e.g., matrix addition and Hadamard product.

(6) Binary operations in which both inputs may be large and which usually require communicating a state of data-dependent size, e.g., matrix-matrix multiplication, which is one of the most expensive LA operations.

Based on our above analysis, we pick one operation from each category for **MAT**, as listed in Table 1. In choosing between operations, we pick those that arise commonly in LA-based ML algorithms. For instance, Frobenius norm in the second category is often used for normalizing the data, while Gramian in the third category arises in OLS.

3.1.2 PIPE: *Pipelines and Decompositions*

LA operations can be composed into pipelines, which are typically steps in a more complex algorithm. Such pipelines present opportunities for *inter-operator optimization*. Perhaps the best known example is matrix chain multiplication, which is analogous to join order optimization in RDBMSs [71]. Consider a simple example: $\mathbf{X_1}\mathbf{X_2}\mathbf{X_3}$, wherein $\mathbf{X_1}$ is $n \times 1$, $\mathbf{X_2}$ is $1 \times n$, and $\mathbf{X_3}$ is also $n \times 1$. The naive left-to-right evaluation plan computes an intermediate matrix of size $O(n^2)$, which could exhaust available memory for large n

Table 1: Tests/settings for Axis 1 (task complexity). Upper case symbols in bold font (e.g., $\mathbf{X}$) represent matrices; lower case symbols in bold font (e.g., $\mathbf{w}$) represent vectors. $\mathbf{X}_{i,j}$ is cell (i,j) of $\mathbf{X}$. $\boldsymbol{\epsilon}$ is the residual vector from OLS.

Test Description	Test Semantics
MAT: Matrix Operators $\sim$ **Vary**: Nodes, Cores, Rows, Sparsity	
Category (1): Matrix Transpose (TRANS)	$\mathbf{X}^T; \mathbf{X}_{i,j}^T = \mathbf{X}_{j,i}$
Category (2): Frobenius Norm (NORM)	$\|\mathbf{X}\|_F = \sqrt{\sum_i \sum_j |\mathbf{X}_{i,j}^2|}$
Category (3): Gram Matrix (GRM)	$\mathbf{X}^T\mathbf{X}; (\mathbf{X}^T\mathbf{X})_{i,j} = \sum_k \mathbf{X}_{k,i} \cdot \mathbf{X}_{k,j}$
Category (4): Matrix-Vector Multiplication (MVM)	$\mathbf{X}\mathbf{w}; (\mathbf{X}\mathbf{w})_i = \sum_k \mathbf{X}_{ik} \cdot \mathbf{w}_k$
Category (5): Matrix Addition (ADD)	$\mathbf{M} + \mathbf{N}; (\mathbf{M} + \mathbf{N})_{i,j} = \mathbf{M}_{i,j} + \mathbf{N}_{i,j}$
Category (6): Matrix Multiplication (GMM)	$\mathbf{M}\mathbf{N}; (\mathbf{M}\mathbf{N})_{i,j} = \sum_k \mathbf{M}_{i,k} \cdot \mathbf{N}_{k,j}$
PIPE: Pipelines and Decompositions $\sim$ **Vary**: Rows	
Multiplication Chain (MMC)	$\mathbf{X_1}\mathbf{X_2}\mathbf{X_3}$
Singular Value Decomposition (SVD)	$\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T \leftarrow \mathbf{X}$
ALG: Bulk LA-based ML Algorithms $\sim$ **Vary**: Nodes, Cores, Rows, Sparsity	
Ordinary Least Squares Regression (OLS)	$(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$
Logistic Regression (LR)	See Algorithm 2
Non-negative Matrix Factorization (NMF)	See Algorithm 3
Heteroscedasticity-Robust Standard Errors (HRSE)	$(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\mathrm{diag}(\boldsymbol{\epsilon}^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$

(e.g., 10 million) and waste runtime. But since matrix multiplication is associative, the right-to-left plan will yield the same result, albeit much faster, since the intermediate matrix is only $O(1)$ in size. For longer chains, one can also have bushy plans with different costs, analogous to bushy plans for multi-table joins. The cost of a plan depends on the dimensions of the base matrices. A "smart" LA system should use metadata to determine the lowest cost plan. Thus, we include this simple pipeline as a test workload. Perhaps surprisingly, most popular scalable LA systems do not optimize this pipeline; the user has to manually fix the multiplication order, which could be tedious in general.

Our second test is singular value decomposition (SVD), which has many applications, including principal component analysis and solving systems of equations. But not all systems support SVD on larger-than-memory data. Thus, we include SVD to serve as a yardstick for the maturity of scalable LA systems.

3.1.3 **ALG**: *Bulk LA-based ML Algorithms*

This is the highest level of task complexity: algorithms expressed as bulk LA scripts. For tractability sake, we include only a handful of popular algorithms that satisfy the following desiderata. First, we should cover a variety of use cases, including regression, classification, feature extraction, and statistical analysis. Second, we should cover a spectrum of computation and communication behaviors. Third, we should cover a spectrum of implementation effort. Fourth, at least some of the compared systems should have "native" non-LA-based implementations of some algorithms (say, using SGD) to let us asses the performance costs or gains of using the LA abstractions. Finally, as a converse, we should have an LA-based algorithm that is *not* widely available as a native implementation to exemplify the utility of scalable LA abstractions, since without such abstractions, users might have to write low-level code to scale such algorithms.

Given our above analysis, we include the following tests. First is ordinary least squares (OLS) for linear regression solved using the normal equations [37]. Second is logistic regression (LR) for binary classification solved using gradient descent [58]. Third is non-negative matrix factoriza-

tion (NMF) for feature extraction [40]. Fourth and final is White's heterscedasticity-robust standard error estimator (HRSE) for OLS, which is commonly used in the domain sciences and enterprises for handling heteroscedasticity [69]. Among these four algorithms, HRSE is *not* available as a native implementation in any of the compared systems (except MADlib). We now briefly explain each algorithm and present their respective LA scripts.

Ordinary Least Squares (OLS). OLS is the most popular algorithm for linear regression [11, 37]. Given the data matrix $\mathbf{X}_{n \times d}$ and target $\mathbf{y}_{n \times 1}$ with values in $\mathbb{R}$, OLS computes the projection of $\boldsymbol{y}$ onto the space spanned by $\boldsymbol{X}$. OLS is trivial to implement in LA systems, since it has a closed form analytical solution. This algorithm stress tests computation of the Gram matrix $\boldsymbol{X}^T\boldsymbol{X}$, a common pattern in ML algorithms. Algorithm 1 presents the LA script for OLS.

Algorithm 1: Ordinary Least Squares (OLS).

Inputs: X, **y**
1 **return** $(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$

Logistic Regression (LR). LR is the most popular ML technique for classification [11]. It is typically solved using iterative optimization algorithms expressed in LA. A simple algorithm is batch gradient descent (BGD); while it is seldom used directly for LR, BGD's data access and communication behaviors are similar to more popular algorithms, such as limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS), that often converge in fewer iterations [46, 58]. The convergence properties of optimization algorithms and ML accuracy are *orthogonal* to the LA system used; thus, we focus only on per-iteration runtimes. This algorithm stress tests matrix-vector multiplication. Algorithm 2 presents the LA script for LR with BGD.

Non-negative Matrix Factorization (NMF). NMF factorizes $\mathbf{X}_{n \times d}$ into two lower rank matrices such that the reconstruction error is minimized. NMF is common for feature extraction, especially on sparse data (e.g., in text

Algorithm 2: Logistic Regression (LR) with BGD.

Inputs: X, y, I: number of iterations, α: step size
1 **for** $i = 1$ to I **do**
2 $\quad \mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \mathbf{X}^T \left(\frac{1}{1+e^{-\mathbf{X}\mathbf{w}}} - \mathbf{y} \right)$
3 **return w**

---

mining). NMF can also use many optimization algorithms; weighted multiplicative updates is popular [22, 49]. Algorithm 3 presents its LA script. This algorithm stress tests multiple LA operations, including matrix-matrix multiplications and multiplication chains.

Algorithm 3: NMF.

Inputs: X, I: number of iterations, r: rank
1 **for** $i = 1$ to I **do**
2 $\quad \mathbf{W} \leftarrow \mathbf{W} \odot ((\mathbf{X}\mathbf{H}^T)/(\mathbf{W}\mathbf{H}\mathbf{H}^T))$
$\quad \mathbf{H} \leftarrow \mathbf{H} \odot ((\mathbf{W}^T\mathbf{X})/(\mathbf{W}^T\mathbf{W}\mathbf{H}))$
3 **return** $(\mathbf{W}, \mathbf{H})$

Heteroscedasticity-Robust Standard Errors (HRSE). Standard errors (SE) of OLS coefficients are used for hypothesis tests about the sign and magnitude of the coefficients in many domain sciences and enterprise settings. This often requires addressing heteroscedasticity–a condition that occurs when the variance of the target is correlated with feature values and may lead to incorrect SEs. To mitigate this issue, the procedure of [69], shown in algorithm 4, is often used. While the LA notation is succinct, expressing this computation in lower-level code could be tedious and painful. In spite of the popularity of such procedures, among the systems compared, only MADlib offers a robust OLS variance estimator natively. This affirms the need for scalable LA abstractions to enable data scientists to scale such LA-based procedures easily. This algorithm also stress tests multiple LA optimizations such as avoiding the materialization of the large (but ultra-sparse) matrix diag($\boldsymbol{\epsilon}^2$).

Algorithm 4: HRSE.

Inputs: X, $\boldsymbol{\epsilon}$: residuals from OLS
1 **return** $(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\text{diag}(\boldsymbol{\epsilon}^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$

---

4. EXPERIMENTAL COMPARISON

We now perform a comprehensive experimental comparison of the performance and scalability of the LA systems discussed in Section 2 using our suite of tests. Due to space constraints, we only discuss a subset of our results here for each task complexity level and present the other results in our technical report [66]. In particular, we focus primarily on the distributed memory setting here, which means most of our discussion centers on pbdR, MADlib, MLlib, and SystemML. In general, the data for our experiments will fit fully in distributed memory and disk spills should not be necessary. For a handful of tests, intermediate results *may* exceed distributed memory depending on the execution plan selected by the system. All the code

Table 2: Software Packages and Versions

Name	Version	Name	Version
Spark	2.2.0	Hadoop	2.7.3
R	3.4.3	pbdR	0.4-2
Python	2.7.12	TensorFlow	1.4.1
NumPy	1.14.0	OpenBLAS	114fc0bae3a
SystemML	d91d24a9fa	MADlib	1.12
Greenplum	5.1.0	SciDB	18.1

for our tests on these systems, as well as our data generation scripts (for synthetic data) and data pre-processing scripts (for Criteo) are available for download at `https://adalabucsd.github.io/slab.html`.

Experimental Environment. All experiments were run on the CloudLab "Clemson" site using the `c6320` instance type [61]. Each physical node has the following hardware specifications: two Intel E5-2683 v3 14-core CPUs (2.00 GHz), 256GB ECC memory, two 1 TB 7.2K RPM 3G SATA HDDs, and a dual-port Intel 10Gbe NIC (X520) network adapter. On top of these nodes, we use OpenStack to create and manage a virtual cluster. Each OpenStack instance corresponds to a physical node and is allocated 24 CPU cores, 200GB RAM and 800GB of disk. The remaining resources are reserved to run OpenStack and cannot be accessed by the benchmarked application. Scripts to create and manage clusters are available on the project GitHub page. Table 2 describes the versions of software packages used.

Methodology. All runs are repeated five times. The first runtime measurement is discarded (for warming caches). The median of the rest is plotted along with error bars depicting the minimum and maximum runtimes. For MLLib, we persist data with `MEMORY_AND_DISK_SER` storage level to minimize the memory footprint of RDDs. Since Spark does not perform computations until the results are needed, we force the computation of specified operations in MLlib by invoking a simple "count" on the result RDD. Similarly, since SystemML eliminates dead code, we insert a break in its program flow (an "if" clause) and print a small number of matrix cells at random from within this break to force it to compute the outputs. We assume that data has been preloaded into the appropriate storage medium for each system (e.g. HDFS or an RDBMS) and do not include data loading times in any of our measurements. But we noticed that when using the GP-load import utility provided by MADlib, importing a 38GB CSV file took about 467 seconds, while uploading the same CSV file to HDFS was a bit faster at about 379 seconds using the HDFS command line utility. In MADlib, we have to write the output matrices to disk as tables. Thus, all runtimes reported for MADlib include this write time, since it is unavoidable. We emphasize that this writing cost is not incurred by the other systems. To simplify installation, SciDB was run from a Docker container. To verify that Docker did not introduce significant overheads, we replicated a distributed benchmark available on the Paradigm4 website and obtained comparable results.

System Configuration Tuning. Almost all compared systems have many tunable configuration parameters that affect performance. Tuning them is non-trivial and requires considerable user time and expertise. We adopted a policy of "reasonable best effort" for tuning based on the best prac-
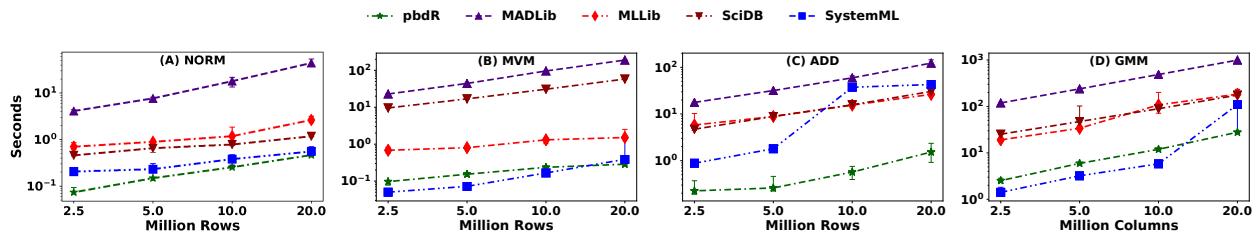
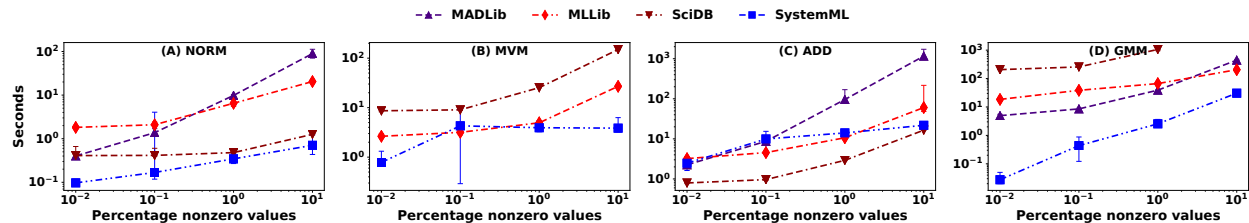**Figure 1:** Multi-Node Dense Data for MAT Varying Rows

Figure 2: Multi-Node Sparse Data for MAT with Varying Sparsity

tices obtained from each system's documentation and online community. We also performed a series of "pre-tests" to tune key parameters, e.g., number of Greenplum segments (for MADlib), number of cores for a Spark Executor (for MLlib), and the LA library to load at runtime (for MLlib and SystemML). There is no universally optimal setting, since it depends on the LA workload run. Thus, for each system, we picked parameters that gave best performance on key LA operations - especially matrix-matrix multiplication. We think this is a reasonable and fair policy, since a typical user is unlikely to spend much time tuning the system compared to running their actual LA workload. Due to space constraints, we discuss more details of our tuning pre-tests in the technical report [66]. We believe our system tuning efforts are reasonable from the standpoint of *typical data scientist users*. While more extensive tuning of a given system might improve its runtimes slightly, that is orthogonal to our larger goal of a unified comparative understanding. We hope our work spurs more interest among the developers and users of such systems to apply our tests for further empirical analysis.

4.1 Results for MAT

Multi-Node - Dense Data - Vary Rows. We fix the number of cluster nodes at 8, the number of columns (d) in matrices at 100, and vary the number of rows (n) for dense data. The data matrix is stored as a CSV file on disk, and varies in size from about 4 GB to 40 GB. For pbdR we generate random data in memory. For general matrix-multiplication in particular, we fix the number of columns in N at 100 and vary the number of rows, with the dimensions of M being the same as that of N^T. Figure 1 presents the results for four of the **MAT** tests. We see that the runtime for each LA operation increases steadily, but the rate of increase is faster for general matrix-multiplication, which is consistent with their communication and computation behaviors explained in Section 3.1.1.

All systems except MADlib finish in under 5s on norm and matrix-vector multiplication. MADlib's LA operations are slower than the others systems', primarily because it has higher per-tuple RDBMS processing overheads and the

I/O time of materializing output tables. SystemML is almost always faster than MLlib, MADlib and SciDB, since SystemML mostly pulls computation into its driver and executes operators in local mode. SystemML's runtimes see a marked rise on ADD at 10 million rows and GMM at 20 million rows; based on the execution logs, we verified that this rise happened because computation was moved to Spark at this data scale. In Spark mode, SystemML exhibits comparable performance to MLlib, which always runs in distributed mode. Thus, SystemML's ability to transparently switch between local and distributed execution based on its data size estimates is effective. Finally, pbdR exhibits strong performance on all four operators in spite of always executing in distributed mode. This underscores both the overhead imposed by the other scalable LA systems and the importance of including a strong baseline system when evaluating performance.

Multi-Node - Sparse Data - Vary Sparsity. We vary the fraction of nonzero entries in $\mathbf{X}$ on the same 8-node cluster. The dimensions are chosen such that $\mathbf{X}$ would be about 100 GB, if materialized as a CSV file. For MVM, $\mathbf{w}$ is dense. We did not find support for sparse data in pbdR. Figure 2 presents results. MADlib's performance generally scales linearly with the density of the matrix. This is because MADlib represents sparse matrices using a tuple of values for row index, column index, and value. In this format, most sparse LA operators can be implemented using join-aggregate queries for which the RDBMS has been extensively optimized. This explains MADlib's stronger showing on these tests relative to the dense matrix operators. MADlib reported an error on MVM because output is returned as an ARRAY rather than a table. At this data scale, the output array exceeds PostgreSQL's hard size limit of 1 GB for arrays. At the time of this writing, this is a serious scalability bottleneck for MADlib that manifests in several other tests. To avoid this issue, users would need to rewrite algorithms in a less intuitive manner to operate on tables in chunks. We note finally that the gap between MLlib and MADlib can be explained, at least in part, by the fact that MLlib always densifies the right operand in GMM, which leads to extra processing overheads.

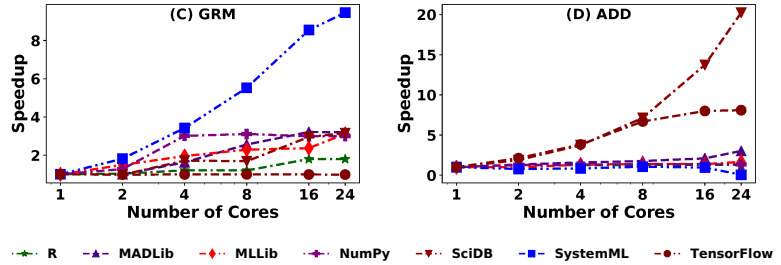| System	GRM (Sec.)	ADD (Sec.)
TensorFlow	10.442	4.230
R	16.787	8.854
NumPy	24.5	9.648
MLlib	18.622	104.556
SystemML	82.990	11.402
SciDB	531.631	1419.915
MADlib	*NA*	1536.775

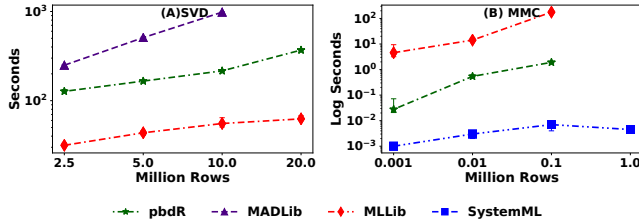**Figure 3:** Single-Node Dense Data for MAT with Varying Cores

Figure 4: Multi-Node Dense Data for PIPE with Varying Rows

Single-Node - Dense Data - Vary Cores. We fix n at 20 million and d at 100, which is about 16GB. We vary the number of cores on a single node to study multicore speedup behaviors. Figure 3 presents the results for two key LA operations. The table shows runtime on a single core to facilitate comparison of absolute performance. We see that the benefit of additional cores typically plateaus at between 4 and 16 cores depending on the system. Speedup behavior varies considerably by both operator and by system. For SciDB, the strong speedup in the ADD operation can be explained by increased parallelism from adding more SciDB instances. For the in-memory systems, multicore speedups are largely dictated by the underlying LA implementation. For example, R's implementation of ADD never uses multiple threads while TensorFlow's does. Such implementation decisions are likely reflective of the different workloads targeted by these systems. R was designed with small to medium size data in mind. In such a context, overhead from setting up threads and copying data may exceed the benefits of parallelism for very simple binary operators like addition. Finally, we note that while TensorFlow's runtimes are generally fastest, it performs all computations using single precision floating point numbers (`float`) by default, while the other systems all use double precision numbers (`double`).

4.2 Results for PIPE

Back to the 8-node setting, we now run matrix multiplication chain (MMC) and SVD on dense data. Figure 4 (A) presents results for SVD. For SVD we fix $d = 100$ and vary n. We compute only the singular vectors corresponding to the ten largest singular values. SystemML is not shown because it does not provide a distributed SVD. SciDB is not shown because its SVD implementation must compute all three factor matrices independently and so a fair comparison is not possible. MLlib's strong performance for this operator is due to a novel algorithm (described in [70]), which is optimized for map-reduce.

Figure 4 (B) presents results for MMC. We omit MADlib and SciDB as the user must specify the multiplication order

explicitly. We see that only SystemML is able to determine the optimal multiplication order and avoid creating the large intermediate matrix. Both pbdR and MLlib have orders of magnitude higher runtimes and eventually crash for even modestly large n. We note, however, that neither pbdR not MLlib claim to be able to detect this optimization; thus, this represents an avenue for improving these systems.

4.3 Results for ALG

Multi-Node - Dense Data - Vary Rows. For the same cluster and data setup as Figure 1, we run the **ALG** tests for dense data. For LR and NMF, the runtimes shown are averaged over three iterations. Figure 5 presents the results. The trends (and explanations) largely follow those of Figure 1, with the relative gap between SystemML and MLlib being larger than for the MAT tests. As with Figure 1, SystemML again performs most computations in the driver program, which explains its high performance, often surpassing pbdR. The jump in runtime for SystemML on HRSE at 10 million rows is again caused by switching to Spark mode. For pbdR, it was necessary to hand-optimize the sparse diagonal matrix multiplication in HRSE to avoid out-of-memory errors. MADlib does not automate composition of LA operators, which forces all intermediate results to be written to disk, leading to high I/O costs and runtimes for iterative algorithms. SciDB does not provide an operator for matrix inversion or solving a linear system. To implement OLS and HRSE on SciDB, we were forced to compute $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ using SVD. Furthermore, to compute the inverse, it is necessary to fully compute $SVD(\boldsymbol{X}^T\boldsymbol{X})$ thrice as the SVD operator provided is strangely only able to return a single matrix in the decomposition per call.

Single-Node - Dense Data - Vary Cores. For the same single-node setup and data scales as Figure 3, we run the **ALG** tests for dense data to study the multicore speedup behaviors of all systems. To obtain tractable runtimes for MADlib, we reduce matrix size to 2.5×10^6. Figure 6 presents the results. Speedup behavior varies substantially by system and algorithm. R is faster than NumPy for LR and HRSE, but not on the others, likely due to the overheads of intermediate data copies for matrix-matrix multiplication [64]. Surprisingly, MLlib is comparable in performance to TensorFlow on OLS and LR. But it is slower on NMF and much slower on HRSE due inefficiency in the method used to compute the dense-diagonal product in HRSE, which we were forced to code by hand. We linked MLlib with a native BLAS library because it yielded substantial performance improvements over the default JVM implementation. For TensorFlow, we used a special "loop" operator provided
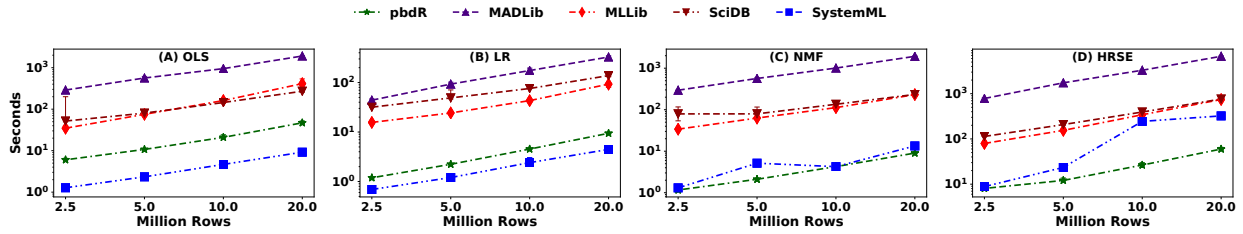
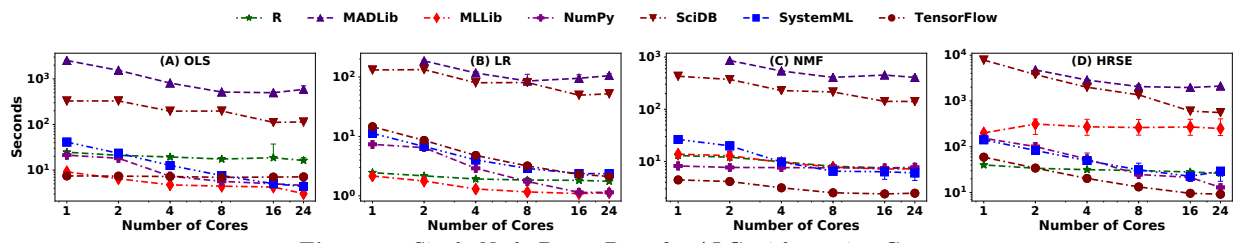**Figure 5:** Multi-Node Dense Data for ALG with varying Rows

Figure 6: Single-Node Dense Data for ALG with varying Cores

by the TF API, since it does not require us to copy data between NumPy and TF at each iteration.

4.4 Scalability on Criteo Datasets

While our above results with synthetic data already give interesting insights about the compared systems, we also want to know how these systems perform on a large real-world dataset for ML analytics. In particular, we are also interested in comparing the native, non-LA based implementations offered by some systems against their own LA-based versions. We use a large public dataset from Criteo, whose full size is 1 TB [5]. Each example is an ad display event. The target is binary, indicating if the ad was clicked or not, but we can treat it as numeric too for our purposes. There are 13 numeric features (integer counts) and 26 categorical features (32-bit hash strings) overall. For the sake of tractability in terms of runtimes, we subsample this dataset. Our sample has 200 million examples and is 50 GB in raw form. We pre-process this version to create two benchmark versions for our comparison: *DenseCriteo* and *SparseCriteo*.

DenseCriteo retains the target and 10 numeric features. Its size as a CSV file is 32 GB. We picked this size to ensure all systems are able to work and at least most of them are able to finish within our timeout constraints. We impute missing values with just zeros, since ML accuracy is orthogonal to our focus. *SparseCriteo* retains the target and 4 categorical variables, which are pre-converted with one-hot encoding [15]. The resulting data matrix has $71,000$ features with a sparsity of 0.0172%. Since Criteo's data usage agreement does not authorize data redistribution, we have released all of our scripts for downloading, pre-processing, cleaning, and obtaining the above two benchmark dataset versions on our project webpage to aid reproducibility.

We compare the scalability of the LA-based and native implementations of OLS and LR for MADlib, MLlib, and SystemML on our cluster by varying the number of cluster nodes from one to eight. We skip GNMF and HRSE because most of these systems do not have native implementations these algorithms. SystemML's native LR and OLS use a hybrid conjugate gradient-trust region optimization algorithm. MLlib's native LR uses L-BFGS, while its native OLS uses IRLS. MADlib's native LR offers a choice between

conjugate gradient, iteratively reweighted least squares, and SGD. We use SGD, since it has the lowest per-iteration runtime. MADlib's native OLS is solved directly as in algorithm 1. Note that these different optimization algorithms have different convergence properties and thus, one will almost surely need different numbers of iterations to reach a similar ML accuracy. But these properties are orthogonal to the scalable LA system used, and thus, orthogonal to our focus. For our experiments, we report the per-iteration runtime of each iterative algorithm, both LA-based and native, averaged from three iterations for a more reliable estimate.

Multi-Node - *DenseCriteo* - Vary Nodes. Figure 7(A) presents the results. We observe three major trends overall. First, across the board, the native implementations of LR and OLS have much lower per-iteration runtimes than the LA-based implementations (except for SystemML on LR). Thus, the scalable LA abstractions, especially on MADlib and MLlib, come at a high performance penalty, which users should be aware of. Second, at the largest setting of 8 nodes, all three systems have comparable performance with their native implementations, while SystemML is the fastest for LA-based implementations. Both of these trends suggest that users might be better off using the native implementations of well-known algorithms such as LR and OLS and resort to the scalable LA abstractions only if absolutely necessary. Third, all systems exhibit near-linear speedups on LR (except LA-based LR in SystemML, which is the fastest), illustrating such speedups are possible at this data scale, but almost none of them exhibit such speed-ups on OLS (except MLlib, which is the slowest). This sub-linearity suggests that communication cost for computing the Gramian in OLS remains a scalability bottleneck. We note that SystemML internally uses a lower number of partitions on the underlying RDD than does MLlib, which limits the degree of parallelism and explains the lack of speedup after two nodes when executing in Spark mode. Increasing the number of RDD partitions increases parallelism but decreases the size of each individual task, potentially reducing throughput.

Multi-Node - *SparseCriteo* - Vary Nodes. Figure 7(B) presents the results. This plot has much fewer lines than Figure 7 (A) due to two reasons. First, the LA-based imple-
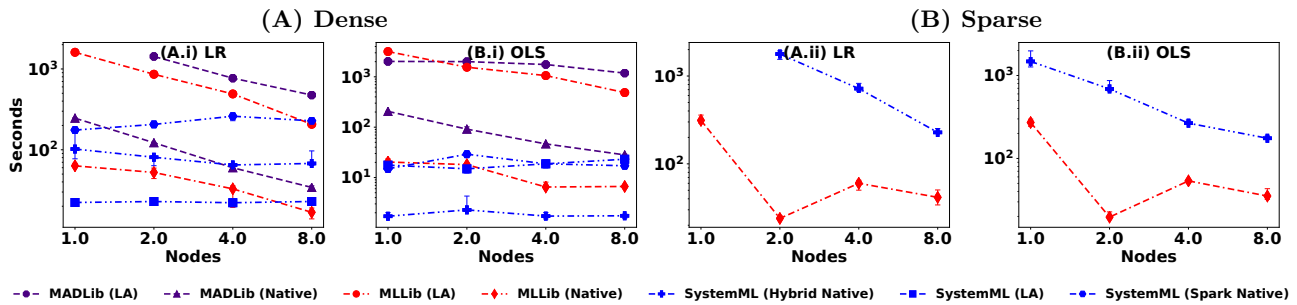
(A) Dense   **(B) Sparse**

Figure 7: Multi-Node *Criteo* for LA-based and native implementations of **LR** and **OLS**

Table 3: Tokens for Implementation of ALG Tests.

System	LR	OLS	NMF	HRSE
pbdR	42	10	49	17
SystemML	40	10	44	32
MLlib	83	26	40	35
MADlib	96	89	191	103
SciDB	109	103	183	138

mentations of both MADlib nor MLlib, as well the native LR and OLS of MADlib were either incapable of running due to hard-coded dimensionality limits, or they crashed or timed out! For instance, the Gramian computation offered by MLlib's `IndexedRowMatrix` datatype has a hard limit of $65,535$ columns, which is lower than the $70,000$ features in *SparseCriteo*. Among the LA-based implementations, only SystemML finished within the timeout but its runtimes were much higher than MLlib's and SystemML's respective native implementations, which obfuscated the trends. The large drop in Spark runtime between 1 and 2 nodes is because Spark must use disk cache in the single node configuration, while the multi-node configurations can use memory only.

5. ANALYSIS AND DISCUSSION

We first discuss the implementation effort needed for the ALG tests. We then provide concrete guidance to practitioners on the strengths and weaknesses of all systems and a summary of the key takeaways from our empirical analysis. Finally, we discuss some open research questions.

5.1 Algorithm Implementation Effort

To evaluate the implementation effort required for writing the **ALG**, one should ideally conduct a user study with real-world data scientists to obtain a thorough picture of system usability. In lieu of such an extensive study, we provide a rough quantification by tabulating the number of "tokens" needed to implement each algorithm in each system in Table 3. We exclude "non-essential" tokens such as calls used to induce computation by Spark.

Unsurprisingly, SystemML and pbdR generally had the most parsimonious code, since their languages have a near-math syntax for LA scripts. In contrast, SciDB generally had the most verbose code mainly due to the need to write driver programs in Python and compute matrix inverses using SVD. MADlib's verbosity stems from its requirement of writing out each matrix operator as a separate PL/SQL call. Finally, MLlib's verbosity is because it does not yet provide a few basic LA operators in its API, which users are forced to write, and because of the need to inject type hints.

5.2 Guidelines for Practitioners

We now summarize each system's strengths and weaknesses. We organize our discussion around the practical difficulties we faced when implementing the ML algorithms in 3.1.3 and specifically address the following questions. How difficult is system setup and configuration tuning? How robust is the support for large sparse data? To what extent is physical data independence supported? Is there support for ad hoc feature engineering/data pre-processing pipelines that precede ML training? Are fault tolerance and automatic disk spills supported? We elide discussion of the single-node tools (R, NumPy, and TensorFlow), since they are generally well known to practitioners.

pbdR. We found tuning pbdR to be simple. Default parameters yielded high performance in most cases. As of this writing, pbdR does not support sparse data. It also has poor support for data I/O; users will likely need to implement custom data readers/writers. It provides a high level of physical data independence; most conventional R scripts can be ported to pbdR with little effort. But pbdR lacks support for complex feature engineering pipelines that Scikit-learn and Spark support. It does not support fault tolerance or automatic disk spills. It also supports only a small subset of R's native ML implementations; users will need to manually re-implement many existing algorithms available in single node R. Overall, pbdR is best suited to users who want to rapidly prototype new LA-based analysis algorithms at scale.

MLlib. We found tuning MLlib to be highly complex. We found that MLlib often crashed or exhibited very poor performance with default tuning settings. Furthermore, we found that tuning settings were often workload specific and could only be determined through a tedious process of trial and error. MLlib supports sparse matrix types, but it sometimes densifies data during execution, degrading performance. Physical data independence is limited as users must manually tune partitioning, caching behavior, and select from a set of matrix data structures supporting inconsistent operators. Moreover, some **ALG** tasks required us to write Scala code to type cast data. The Spark ecosystem offers excellent support for data pre-processing and feature engineering pipelines at scale, fault tolerance, and disk spills in many settings. MLlib's native ML implementations were typically faster than LA-based implementations, but we encountered scalability issues with its matrix types, especially for sparse data. Overall, MLlib is best suited to users who require robust support for complex data pre-processing but are happy with canned ML algorithm implementations.

SystemML. We found tuning SystemML to be simpler than MLlib, since it abstracts away data layout and caching decisions. SystemML has generally good support for sparse data, but the metadata overheads for empty data blocks are non-trivial and caused performance and scalability issues in some cases. SystemML offers APIs to interpolate between Spark and DML. This allows users to leverage the robust support for data pre-processing offered by the Spark ecosystem, while employing DML's higher level LA abstraction. Since SystemML uses Spark RDDs, it inherits Spark's fault tolerance and disk spill capabilities. SystemML's native ML implementations typically had higher runtimes (per iteration) than the LA-based versions in our **ALG** tests, but on datasets with low d, its native ML implementations can typically converge in fewer iterations. Overall, SystemML is best suited to users who want to deploy custom LA-based ML algorithms in a production environment where robust feature pre-processing pipelines and fault-tolerance are essential.

MADlib. We found tuning MADlib on Greenplum to be relatively simple but laborious. There is not much concrete advice on tuning the number of Greenplum segments per node [10]. Determining the optimal number of segments required manually re-initializing and loading the database which was labor-intensive. MADlib has generally good support for sparse data and was competitive on the **MAT** tests with sparse data. However, it suffers from a serious scalability bottleneck due to a hard limit of 1 GB imposed on the maximum size of an array by PostgreSQL. MADlib's use of an RDBMS gives it a reasonable level of physical data independence, since data partitioning and caching are abstracted away. However, interleaving table creation in SQL and LA computations might be unintuitive for many ML-oriented data scientists. The RDBMS offers excellent support for data pre-processing using SQL, but flexibility for writing feature engineering pipelines is poorer than Spark. Greenplum supports fault tolerance and automates disk spills (except for the array type). MADlib's native ML implementations were typically significantly faster than LA-based implementations and were competitive with the other systems compared. Overall, MADlib is well suited to users with large RDBMS-resident datasets and who are happy with canned ML algorithm implementations.

SciDB. Similar to MADlib, tuning SciDB is simple but tedious. SciDB requires tuning (at a minimum) the number of DB instances used and the partition size of matrices. While it has support for sparse matrices, it densifies them when computing SVD and GMM. Surprisingly, GMM and SVD are the only major LA operators provided by SciDB in the distributed setting. This forces users to manually implement other LA operators using these building blocks. SciDB does support automatic disk spills. The community edition of SciDB does not provide any "canned" ML algorithms. Overall, while SciDB is well suited to users that want to manage large multi-dimensional array data, it requires a lot of implementation effort on the part of the user when performing non-trivial LA-based ML algorithms.

5.3 Key Takeaways

We now provide a brief summary of the key insights on system design we gleaned from our experimental analysis and experience using these tools.

(1) Transparently switching between execution in local and distributed mode improves performance and reduces user effort. Recent surveys of data scientists show that a majority work with datasets that could fit on a beefy commodity node's memory [12, 24]. By providing APIs that work mostly seamlessly in both local and distributed mode, SystemML attained good performance on smaller datasets without requiring us to re-implement code for scaling it to large datasets.

(2) Automatically detecting LA pipeline optimizations such as reordering matrix multiplication chains and multiplication of a dense matrix with a diagonal matrix (as in HRSE) yields performance benefits and reduces user effort. Once again, only SystemML detected such optimization opportunities automatically. The other systems required us to hand-optimize multiplication orders, which often required trudging through their documentation.

(3) Intermediate results should not be needlessly materialized and computations should be pipelined. MADlib's performance suffered from the need to write all intermediate results to disk only to read them again in the next line of code. This cripples the RDBMS's ability to perform any LA-specific *inter-operator* optimizations.

(4) Abstractions that purport to support physical data independence must lead to real reductions in user coding effort. SystemML and pbdR both provide strong physical data independence by offering only a single logical "matrix" data type. In contrast, MLlib has too many matrix types with inconsistent APIs that yield dramatically different performance depending on the LA operation. This forces users to experiment with different matrix types and to manually implement LA operators not supported on a chosen type.

5.4 Open Research Questions

Extrapolating from our points above, we identify a few key gaps that require more research from the data systems community. We discuss four major groups of issues.

Better Support for Large Sparse Data. None of compared systems offered the coveted combination of strong physical data independence and high performance at scale for large sparse data, which are increasingly common in ML applications. MLlib and TensorFlow have relatively poor physical data independence, and while pbdR is a strong baseline for dense data, it lacks support for sparse data. SystemML offers perhaps the best combination so far for large sparse data, but it too has high metadata overhead in some cases. Overall, more work is needed for fully supporting sparse data for efficient and scalable LA, including optimizing data layout, staging computations and communication, and managing caching for LA scripts. This relates to the challenging problem of predicting sparsity of intermediates.

Auto-tuning Data Layout and System Parameters. The promise of physical data independence means users should not have to expend much effort for tuning data layout and system parameters. Yet, most systems required us to tweak one or more of such parameters; some just crashed without such tuning. For instance, MLlib required manually tuning partitioning and caching commands, while MADlib required several data layout decisions for interleaving LA scripts with table creation in SQL. Such low-level decisions

are likely to be unintuitive for statistical or ML-oriented users. MLlib and TensorFlow also require lower-level programming skills in Python (or Scala for MLlib) to resolve such issues. SystemML offers perhaps the most automation of such decisions, but its dependence on Spark necessitates some memory-related tuning. Overall, more work is still needed to achieve true physical data independence for scalable LA. Extending the lessons of auto-tuning from the RDBMS and MapReduce worlds (e.g., [42, 67]) to scalable LA is another avenue for new research.

Multi-node "COST" and Parallelism Models. A surprising takeaway relates to the bedrock of database systems: multi-node data parallelism. Conventional wisdom has that using many cheap nodes is likely better than fewer expensive beefy nodes, even for dozens of GBs. But as our Criteo results show, the speedup curves sometimes flatten quickly, at least for some of the compared algorithm implementations. This issue is the multi-node version of the "COST" factor in [53]. It is not clear if this issue will be mitigated at larger scales (TBs or PBs), since that would require even more nodes to get more total memory (at least for SystemML and MLlib). In turn, that could raise communication costs for LA operations, as well as associated metadata and query processing overheads. Furthermore, recent surveys of data science practitioners show that about 70% analyze datasets under only 100 GB [12, 24]. Thus, while faster multi-node data-parallel LA implementations are useful, we think a more pressing research challenge is to support transparent scalability for hybrid *data and task parallelism*. Such support is useful for crucial meta-level ML model selection tasks such as hyper-parameter tuning [24, 44, 45, 63]. A recent Spark-TensorFlow integration does support task-parallel hyperparameter-tuning for neural networks but only for small datasets that can be broadcasted [6]. SystemML also provides framework for multi-node task parallelism, but does not automate hyperparameter tuning [26]. A related research question is determining the optimal cluster size for a given dataset and LA script, similar to how [63] sizes clusters for a few specific ML algorithms.

Including ML Accuracy for Evaluation. The faster performance of MLlib's and MADlib's native ML implementations compared to their LA-based versions suggests that including ML accuracy as a criterion could substantially alter the relative performance landscape. But it is non-trivial to standardize comparisons of both runtimes and ML accuracy *simultaneously* because the latter is inextricably tied to hyper-parameter tuning, which is usually a non-smooth and non-convex meta-level optimization problem [63]. A given system will likely fall on multiple points on the accuracy-runtime *Pareto frontier* for different datasets, ML algorithms, and hyperparameter-tuning choices, which makes apples-to-apples comparisons between systems hard. Moreover, different implementations of the same ML task could have differing hyper-parameters. For instance, how does one standardize how hyperparameter-tuning should be done when comparing, say, MADlib's SGD-based LR with SystemML's native LR? Should sub-sampling be allowed to enable more extensive hyper-parameter tuning? We leave such questions to future work but note that there is growing interest in Pareto frontier-based benchmarks, at least for deep learning tasks, although they do not yet account for the costs of hyper-parameter tuning [31].

6. OTHER RELATED WORK

Benchmarks of LA Packages. LA packages such as BLAS, LAPACK, ScaLAPACK, and Eigen have been extensively benchmarked [21, 23, 34, 35, 48, 57]. For instance, the LINPACK benchmark focuses on the efficiency (measured in MFLOPs per second) of solving a system of linear equations [33, 47]. BLAS has a long history of development, while LAPACK and ScaLAPACK build upon BLAS. Numerous scalable LA tools were subsequently built and benchmarked, including multicore-specific implementations in PLASMA [47] and numerous implementations to exploit GPUs and other hardware accelerators. There is a long line of work by the high-performance computing and supercomputing community on building and benchmarking distributed LA frameworks, primarily for scientific computing applications [36, 47]. Such implementations typically rely on custom compilers and communication frameworks. In contrast, our works focuses on a comprehensive comparative evaluation of recent scalable LA systems built on top of standard data systems (MADlib, MLlib, SystemML) along with TensorFlow on an even footing. We also include R, NumPy, and pbdR/ScaLAPACK as strong baselines. Recent work has also profiled R to understand its memory usage and execution overheads but their goal was to improve single-node R, not comparing scalable LA tools [64].

Comparisons of Analytics Systems. MLlib, SystemML, MADlib, and TensorFlow were released in their current form only in the last four years. While their reference publications show several results [20, 25, 41, 55, 65], there is no uniformity in the workloads, data scales, and computational environments studied. Our work fills this crucial gap in the literature by systematically comparing them on an even footing. Recent work has also evaluated the performance of Apache Mahout Samsara but only for a few operations [62]. [54] compared SciDB [27], Myria [68], Spark, and TensorFlow for a specific scientific image processing task but not general LA or ML workloads. [28] compare Spark, GraphLab [50], SimSQL [29], and Giraph [3] for Bayesian ML models implemented using lower-level abstractions of such systems; they did not evaluate general LA or LA-based ML workloads. [24] compared MLlib's native ML implementations with single-node tools such as Vowpal Wabbit [19] to understand the COST factor for distributed ML. One of our findings is similar in spirit but our work is more general, since we cover scalable LA and LA-based data analytics workloads, not just a few specific ML algorithms. We also include MADlib, SystemML, and the oft-ignored pbdR/ScaLAPACK in our study. Finally, [31] introduce new benchmark criteria for comparing deep learning tools/models on both accuracy and monetary cost for some image and text prediction tasks. However, it does not yet cover bulk LA workloads or structured data analytics. Thus, overall, all these prior benchmarking efforts are largely orthogonal to our work.

7. REFERENCES

[1] 2017 Big Data Analytics Market Survey Summary. https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/#4b513fce39a1. Accessed September, 2018.

[2] Amazon Web Services ML. https://aws.amazon.com/machine-learning/. Accessed September, 2018.

[3] Apache Giraph. http://giraph.apache.org. Accessed September, 2018.

[4] Apache SystemML Webpage. https://systemml.apache.org/. Accessed September, 2018.

[5] Criteo Terrabyte Click Logs. http://labs.criteo.com/2013/12/download-terabyte-click-logs/. Accessed September, 2018.

[6] Deep Learning with Apache Spark and TensorFlow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html. Accessed September, 2018.

[7] Distributed TensorFlow Webpage. https://www.tensorflow.org/deploy/distributed. Accessed September, 2018.

[8] Gartner Report on Analytics. gartner.com/it/page.jsp?id=1971516. Accessed September, 2018.

[9] Google Cloud ML Engine. https://cloud.google.com/ml-engine/. Accessed September, 2018.

[10] Greenplum Tuning Guidelines. https://gpdb.docs.pivotal.io/530/admin_guide/intro/arch_overview.html#arch_segments. Accessed September, 2018.

[11] Kaggle Survey: The State of Data Science and ML. https://www.kaggle.com/surveys/2017. Accessed September, 2018.

[12] KDNuggets Poll of Data Scientists for Largest Dataset Analyzed. https://www.kdnuggets.com/2016/11/poll-results-largest-dataset-analyzed.html. Accessed September, 2018.

[13] Microsoft Azure ML. https://azure.microsoft.com/en-us/services/machine-learning-studio/. Accessed September, 2018.

[14] Microsoft Revolution R. http://blog.revolutionanalytics.com/2016/01/microsoft-r-open.html. Accessed September, 2018.

[15] One-Hot Encoding Example in Scikit-learn. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. Accessed September, 2018.

[16] Oracle R Enterprise. www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html. Accessed September, 2018.

[17] Project R Webpage. r-project.org. Accessed September, 2018.

[18] SparkR Webpage. spark.apache.org/R. Accessed September, 2018.

[19] Vowpal Wabbit. https://github.com/JohnLangford/vowpal_wabbit/wiki. Accessed September, 2018.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2016.

[21] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[22] M. W. Berry, M. Browne, A. N. Langville, V. P. Pauca, and R. J. Plemmons. Algorithms and applications for approximate nonnegative matrix factorization. *Computational Statistics & Data Analysis*, 52(1):155–173, 2007.

[23] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[24] C. Boden, T. Rabl, and V. Markl. Distributed Machine Learning - but at what COST? In *Machine Learning Systems Workshop at the 2017 Conference on Neural Information Processing Systems*, 2017.

[25] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda. SystemML: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.

[26] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in systemml. *PVLDB*, 7(7):553–564, 2014.

[27] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[28] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1371–1382, New York, NY, USA, 2014. ACM.

[29] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 637–648, New York, NY, USA, 2013. ACM.

[30] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *PVLDB*, 10(11):1214–1225, 2017.

[31] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and

M. Zaharia. DAWNBench: An end-to-end deep learning benchmark and competition. In *Machine Learning Systems Workshop at the 2017 Conference on Neural Information Processing Systems*, 2017.

[32] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of fortran basic linear algebra subprograms: Model implementation and test programs. Technical report, Argonne National Lab., IL (USA), 1987.

[33] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[35] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[36] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[37] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[38] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[39] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on mapreduce. In *2011 IEEE 27th International Conference on Data Engineering*, pages 231–242, 2011.

[40] N. Gillis. Introduction to nonnegative matrix factorization. *SIAG/OPT Views and News*, 25(1):7–16, 2017.

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

[42] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research*, pages 261–272, 2011.

[43] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, New York, NY, USA, 2017. ACM.

[44] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *Proceedings of the Sixth Biennial Conference on Innovative Data Systems Research*, 2013.

[45] A. Kumar, R. McCann, J. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *ACM SIGMOD Record*, 44(4):17–22, 2016.

[46] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1969–1984, New York, NY, USA, 2015. ACM.

[47] J. Kurzak, D. A. Bader, and J. Dongarra. *Scientific Computing with Multicore and Accelerators*. CRC Press, Inc., Boca Raton, FL, USA, 2010.

[48] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[49] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788, 1999.

[50] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, pages 340–349, 2010.

[51] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *2017 IEEE 33rd International Conference on Data Engineering*, pages 523–534, 2017.

[52] MADLib Development Team. MADLib User Documentation. `http://madlib.incubator.apache.org/docs/latest/index.html`. Accessed September, 2018.

[53] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems*. USENIX Association, 2015.

[54] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad. Comparative evaluation of big-data systems on scientific image analytics workloads. *PVLDB*, 10(11):1226–1237, 2017.

[55] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. MLlib: Machine learning in

Apache Spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[56] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, first edition, 1997.

[57] MKL Development Team. Intel math kernel library developer reference. `https://software.intel.com/en-us/articles/mkl-reference-manual`. Accessed September, 2018.

[58] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[59] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. *Programming with Big Data in R*, 2012. `http://r-pbd.org/`, Accessed September, 2018.

[60] N. Pansare, M. Dusenberry, N. Jindal, M. Boehm, B. Reinwald, and P. Sen. Deep learning with Apache SystemML. In *SysML*, 2018.

[61] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[62] S. Schelter, A. Palumbo, S. Quinn, S. Marthi, and A. Musselman. Samsara: Declarative machine learning on distributed dataflow systems. In *Machine Learning Systems Workshop at the 2016 Conference on Neural Information Processing Systems*, 2016.

[63] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 368–380. ACM, 2015.

[64] S. Sridharan and J. M. Patel. Profiling R on a contemporary processor. *PVLDB*, 8(2):173–184, 2014.

[65] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 177–188, New York, NY, USA, 2014. ACM.

[66] A. Thomas and A. Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics - technical report. `https://adalabucsd.github.io/papers/TR_2018_SLAB.pdf`.

[67] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, SIGMOD'17, pages 1009–1024, New York, NY, USA, 2017. ACM.

[68] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[69] H. White. Using least squares to approximate unknown regression functions. In *International Economic Review*, pages 149–170. JSTOR, 1980.

[70] R. B. Zadeh and G. Carlsson. Dimension independent matrix square using mapreduce. *arXiv preprint arXiv:1304.1467*, 2013.

[71] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *2010 IEEE 26th International Conference on Data Engineering*, pages 1157–1160, 2010.