A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics [Experiments and Analysis]

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 users in enterprise and domain scientific settings. To mitigate this issue, a new breed of systems expose high-level bulk *linear algebra* (LA) primitives, which scale to large data. 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 providing a unified comparative assessment 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, and TensorFlow using both synthetic datasets and a large real-world dataset. Our study has revealed several scalability bottlenecks, unusual performance trends, and even bugs in some of these 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 [6, 22]. Consequently, both the database and cloud service industries are building analytics systems that aim to make it easier to integrate ML algorithms and frameworks such as R with data platforms [1,7,11,12,14,36,55]. 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 datasets. Such users typically write their algorithms using *linear algebra* (LA) notation and are used to tools such as R, SAS, and Matlab [9, 15]. LA is a succinct and elegant mathematical language to express matrix transformations for a wide variety of statistical and ML algorithms [36, 42]. 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 easily scale their LA-based algorithms to distributed memory-based or disk-based datasets without needing to manually handle data distribution, communication, fault tolerance, etc. Thus, they offer a measure of *physical data independence* for LA-based analytics. Examples include RIOT [86], EMC's MADlib [55], Oracle R Enterprise [14], IBM's SystemML [36], Microsoft's Revolution R [12], Spark MLlib [71] and SparkR [16], Apache Mahout Samsara [76], LA on SimSQL [67], and Google's TensorFlow [31]. Most of these systems expose an LA-based language or API embedded in Python, Scala, or SQL as their front-end. A key point to note is that by "LA-based algorithms," such systems mean algorithms whose data-intensive computations are *bulk* LA operations over the whole dataset, not mini-batch sampling-based access, which is needed for training neural networks [49, 64]. Thus, except for TensorFlow, these systems do not offer much support for arbitrary neural networks (yet). Nevertheless, bulk LA-based statistical data analysis and LA-based ML algorithms remain a critical use case for structured data analytics, especially in enterprise settings [9].

While the recent flurry of activity on scalable LA systems has led to several interesting research papers and open-source systems, a fundamental practical question remains largely unanswered: *From a comparative standpoint, just how effective and efficient are such systems for scalable LA and LA-based ML?* By effectiveness, we mean how "easy" such systems are to use in terms of the LOC needed to express algorithms using their LA abstractions, as well as their setup logistics. By efficiency, we mean how "fast" such systems run in different settings, especially the increasingly popular distributed memory setting. Different systems have different assumptions and restrictions 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 settings where they perform well, leaving users in the dark on where they may not perform so well. Additionally, most of these systems were not compared against strong baselines such as pbdR (powered by ScaLAPACK) [74]. 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.

To fill the above gap, in this work, we introduce a common suite of tests for scalable LA systems and perform a comprehensive empirical comparison of a few popular and state-of-the-art scalable 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 [32,34,45,46,63,72]. But unlike such previous benchmark comparisons, which focused only on simple LA operations, we want to evaluate the entire hierarchy of task complexity for LA-based data analytics. Thus, we delineate three orthogonal axes of interest: *task complexity*, *data scale*, and *computational environment*. For data scale, we vary both the number of examples and sparsity of the data matrix. For computational environment, we focus on commodity clusters and vary both the number of CPU cores and number of nodes.

As 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 in our study: MADlib, MLlib, SystemML, pBDR/ScaLAPACK, TensorFlow, R, and NumPy. We picked them primarily due to their popularity, but also because they are open source and have active user communities online. Our benchmark comparison of the above systems 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 of the compared systems. For instance, we find that tuning memory-related parameters correctly is crucial to extract good performance from the Spark-based systems (MLlib and SystemML), but the optimal settings are LA task-specific, making it a non-trivial task for data scientists. The 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 quite competitive for LA operations on sparse data. Even more surprisingly, pBDR/ScaLAPACK outperformed all the other systems in almost all cases on dense data. However, taking both performance at scale and physical data independence into account, we find that SystemML offers perhaps the best combination for scalable LA-based analytics today.

Based on our extensive experimental study, we identify and summarize the strengths and weaknesses of the compared scalable LA systems to help practitioners decide which systems best fit 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 not comprehensive in almost all of the compared systems. This is a pressing issue because of the ubiquity of large-domain categorical features in ML-powered applications. We also find that in many cases, the compared systems exhibit sub-linear multi-node speedups, which throws into question the utility of data parallelism for scalable LA in commodity cluster environments. We posit that flexible hybrid parallelism models integrating data and task parallelism within such systems could help make them more useful for crucial meta-level ML tasks such as hyper-parameter tuning [60]. But creating such capabilities requires tackling challenging research problems at the intersection of data management, distributed systems, and ML.

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 pick a suite of tests spanning the axes of task complexity, data scale, and computational environment.
- Using our tests, we perform a comprehensive empirical comparison of the scalability and performance of MADlib, MLlib, SystemML, and ScaLAPACK in the distributed memory setting, along with TensorFlow, R, and NumPy in the single-node setting. Apart from synthetic data of various scales, we create two benchmark versions of a large real-world dataset from Criteo.
- 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 repeatability 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, including overviews of all the LA systems we compare. Section 3 explains our suite of tests in detail. Section 4 presents our comprehensive 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

The 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 [73], non-negative matrix factorization, k-means clustering, and more [42,49,51]. Such algorithms are commonly used in domains ranging from social sciences and physical sciences to enterprise analytics and Web click log analytics.

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 (RA). Different systems take different routes: some layer LA as an abstraction on top of an 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, especially at scale, using a carefully designed suite of tests at multiple levels of abstraction.

2.2 Overview of Compared Systems

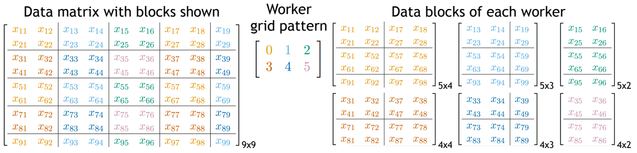
We present a brief overview of the systems we compare empirically, starting with why picked them specifically.

Rationale for Systems Picked. We restrict our empirical comparison 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 SciDB). 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 perhaps the most popular tools [9], with TensorFlow being a new contender. For the latter, the primary transparently scalable LA systems are pBDR, powered by ScaLAPACK [34,74], the distributed RDBMS-based MADlib [55], and Spark-based MLlib (or SparkML) [71] and SystemML [36]. TensorFlow does not support bulk LA in the distributed setting [5]; thus, we skip it for the distributed setting.

R and NumPy. R and Python's NumPy stack treat matrices as first class citizens and provide a rich set of built-in LA operations and algorithms. Most LA operations in R and NumPy are just thin wrappers around highly optimized LAPACK and BLAS routines [32,43]. R and Python also provide robust visualization/plotting libraries (e.g., ggplot and matplotlib) and are Turing-complete. Since R's syntax is close to math notation, it is especially popular among data scientists with a statistics background and in the domain sciences. In fact, the open source repository CRAN contains numerous R libraries contributed by such researchers and practitioners [20]. NumPy, in contrast, is typically more popular among data scientists with a CS background [9]; it uses a function call-oriented syntax. Both R and Python are interpreted and dynamically typed, which makes holistic optimization of LA scripts challenging.

pBDR (ScaLAPACK). ScaLAPACK extends LAPACK to the distributed memory setting by re-implementing many LA operations and algorithms using a block partitioning scheme to distribute data matrices [34]. It follows the "same program multiple data" paradigm in which a single logical program is executed by multiple workers (each worker corresponds to one 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. Figure 1 gives an example of the block-cycling partitioning scheme from [77] for a 9 × 9 matrix with a grid of 6 workers and blocks of size 2 × 2.

Figure 1: Block-cyclic matrix partitioning [77].



In general, a skewed allocation can cause performance issues. Interprocess communication is handled by the BLACS library with an LA-specific message passing API. Overall, both ScaLAPACK and BLACS are low-level libraries that require knowledge of C, FORTRAN, and the intricacies of parallel computing. Thankfully, the "Programming with Big Data in R" (pbdR) library provides higher level R interfaces to ScaLAPACK and OpenMPI [74]. The "distributed matrix" package in pbdR overloads several built-in LA operations in R to enable transparent distributed execution. However, unlike regular R scripts, which can run interactively in a REPL, pbdR compiles programs into OpenMPI batch jobs that are then submitted for execution [50].

MADlib. MADLib is a library that implements both LA primitives and popular ML algorithms over PostgreSQL and the parallel RDBMS Greenplum [55]. Dense matrices are stored as tables with two attributes: an integer row number (the key) and a value attribute that uses the abstract data type `ARRAY`. Sparse matrices are stored as tables with three attributes: row number, column number, and cell value. Thus, MADlib expresses many LA operations directly in SQL and exploits the RDBMS for for memory management and scalability. Low-level in-memory LA operations such as inner products exploit Eigen [53]. To write LA scripts, one has to write SQL queries invoking MADlib's LA routines. Some LA operations such as certain matrix decompositions, however, require the dataset to fit entirely in single-node memory [68]. Related to MADlib is RIOT-DB [86], which avoids SQL as a front-end and opts for the so-called "query generation" approach [59]. A user writes an LA script in R using RIOT-DB's datatypes, which is then translated and optimized (via lazy evaluation) to produce several (procedural) SQL queries [86]. Also related is the recent SimSQL [67], which relies on custom user-defined datatypes for block-partitioned matrices and custom user-defined functions that implement LA operations.

Spark MLlib/SparkML. MLlib (and the newer SparkML) are libraries that provide some LA primitives and popular ML algorithms over Spark. We focus on MLlib, since it is popular among enterprise users [22], and since SparkML does not yet support distributed matrices. Apart from the *LocalMatrix* datatype for small data, MLlib offers three main (physical) datatypes for distributed matrices targeting different data access patterns. (1) *DistributedRowMatrix* (DRM), an RDD with rows of a logical matrix stored using a *LocalVector* datatype. (2) *CoordinateMatrix* (CM), an RDD with tripes of row number, column number, and data value (like MADlib's sparse matrix table). (3) *BlockMatrix* (BM), an RDD of matrix blocks stored using *LocalMatrix*.

DRM supports multiplication with a *LocalMatrix* but not fully distributed matrix multiplication. DRM also supports scalable matrix decompositions such as SVD and QR. CM supports no meaningful scalable LA operations except transpose. Users have to cast it to another distributed matrix type. Sparsity is preserved after casting. BM is the only type that supports fully distributed matrix multiplication. The underlying LA operations over local datatypes are implemented using the Breeze library in Scala [54]. MLlib's datatypes do not yet support many basic LA operations, including scalar-matrix multiplication, norms, and Hadamard product; users have to implement these using RDD operations. Also, the three distributed matrix types are not consistent in the set of LA operations they support, e.g., BM supports transpose but DRM does not, while DRM supports multiplication by a *LocalMatrix* but BM does not.

SystemML on Spark. Introduced for Hadoop and then ported to Spark, SystemML is perhaps the most mature scalable LA system [36]. We focus on the recommended Spark version [3]. SystemML offers a "declarative" language named DML with R-like syntax to express LA scripts (there are also APIs in Python and Scala). DML offers full physical data independence, i.e., users do not decide data layout formats or low-level execution details. SystemML stores matrices in custom binary formats using Spark RDDs, in particular, as block-partitioned matrices with each block stored in a tuple.

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 first level of this translation produces a DAG of so-called "high level operations" (HOPs), which represent basic LA operations. Each HOP is associated with one or more physical execution plans called "low-level operations" (LOPs), which are optimized for data and system characteristics. A HOP-DAG is converted to a LOP-DAG based on both rules and cost-based optimizations to minimize runtimes under memory constraints. LOPs are executed as either RDD operations or in-memory computations in the driver program. SystemML also includes more advanced optimizations such as dead code elimination and operation fusion to reduce data access costs.

TensorFlow. TensorFlow (TF) is a framework for expressing ML algorithms, especially neural networks [31]. 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. Models in TF are expressed as "computational graphs" in which nodes represent operations over multi-dimensional arrays ("tensors") and edges represent dataflow. A TF program has two stages. First, the computational graph is specified and placed on available compute devices. Then, a node is "run" (not necessarily a terminal node), with its inputs made available. Separating these stages enables TF to use lazy evaluation to compile the graph and apply some holistic optimizations within Python's interpreted environment.

TF's LA API offers a set of high-level routines ("operators"), each with one or more physical implementations ("kernels") for specific compute devices. TF supports many device backends, including CPUs, GPUs, and Android smart phones. User can assign different operations to different devices; if such assignments are not specified, TF optimizes the placement to reduce data movement and runtimes. TF also detects and removes redundant computations, but overall, its LA-specific optimizations are not yet as extensive as SystemML's. Also, TF does not yet offer distributed bulk LA operations. Thus, we consider TF a single-node tool. TF is under intense development, with optimizations such as XLA [18] and new extensions being introduced. We refer the interested reader to their webpage for the latest [19].

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 and derived LA operations over the data matrix. Such operations 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 different systems perform at different data scales. For the sake of simplicity and uniform treatment across all systems, we assume the data matrix X consists only of numeric features (categorical features are assumed to be pre-converted using one-hot encoding to obtain sparse 0/1 vectors [13]). There are three key settings: number of rows and number of columns, denoted **D.R** and **D.C** respectively, and the fraction of non-zero cells, also called *sparsity*, denoted **D.S**. This is summarized in Table 2. Note that **D.R** and **D.C** govern the shape of X. We primarily focus on varying **D.R** and **D.S**; for tests that use square matrices as input, we set **D.R** = **D.C**.

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, denoted **E.C** and **E.N** respectively (also shown in Table 2). While this is a popular setting for many enterprises and domain scientists, we note that GPUs, FPGAs, and custom ASICs

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

Test Level/Name	Test Description	Scale Factors Varied
MAT	**Matrix Operations**	All
MAT.1: Matrix Transpose (TRANS)	$\mathbf{X}^T; \mathbf{X}^T_{i,j} = \mathbf{X}_{j,i}$	
MAT.2: Frobenius Norm (NORM)	$\|\mathbf{X}\|_F = \sqrt{\sum_i \sum_j \lvert\mathbf{X}^2_{i,j}\rvert}$	
MAT.3: Gramian Matrix (GRM)	$\mathbf{X}^T\mathbf{X}; (\mathbf{X}^T\mathbf{X})_{i,j} = \sum_k \mathbf{X}_{k,i} \cdot \mathbf{X}_{k,j}$	
MAT.4: Matrix-Vector Multiplication (MVM)	$\mathbf{X}\mathbf{w}; (\mathbf{X}\mathbf{w})_i = \sum_k \mathbf{X}_{ik} \cdot \mathbf{w}_k$	
MAT.5: Matrix Addition (ADD)	$\mathbf{M} + \mathbf{N}; (\mathbf{M} + \mathbf{N})_{i,j} = \mathbf{M}_{i,j} + \mathbf{N}_{i,j}$	
MAT.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**	**D.R = D.C, D.S**
PIPE.1: Multiplication Chain (MMC)	$\mathbf{X_1 X_2 X_3}$	
PIPE.2: Singular Value Decomposition (SVD)	$\mathbf{U\Sigma V}^T \leftarrow \mathbf{X}$	
ALG	**Bulk LA-based ML Algorithms**	**D.R, D.S, C.C, C.N**
ALG.1: Ordinary Least Squares (OLS) Regression	$(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$	
ALG.2: Logistic Regression (LR)	See Algorithm 2	
ALG.3: Non-negative Matrix Factorization (NMF)	See Algorithm 3	
ALG.4: Heteroscedasticity-Robust Std Errors (HRSE)	$(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\text{diag}(\boldsymbol{\epsilon}^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$	

Table 2: Settings varied along Axes 2 and 3.

Factor Name	Factor Description
D:	Data Scale Factors
D.R	n, number of rows of **X**
D.C	d, number of columns of **X**
D.S	Sparsity of **X**
C:	Computational Scale Factors
C.C	Number of CPU cores
C.N	Number of worker nodes

(e.g., TPUs [57]) 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 (cores/nodes) in a pre-specified way (row-wise, column-wise, or block-partitioned). We observe at least six major categories.

First are 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. Second are 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. Third are 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. Fourth are binary operations in which one of the inputs typically fits in single-node memory, e.g., matrix-vector multiplication. Fifth are binary operations in which both inputs are larger than single-node memory but proper co-partitioning can avoid communication, e.g., matrix addition and Hadamard product. The sixth and final category we consider are 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 [85]. Consider a simple example: $\mathbf{X_1X_2X_3}$, 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 (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 the metadata to determine the lowest cost plan. Thus, we include this simple pipeline as test **PIPE.1**. 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 homogeneous systems of equations. But not all systems support SVD on larger-than-memory data. Thus, we include SVD as test **PIPE.2** 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 want to 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 (in terms of LOC). Fourth, at least some of the compared systems should have "native" non-LA-based implementations of some algorithms, e.g., 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 such abstractions to scalable data analysis. Without such abstractions, users might have to write low-level code to scale such algorithms.

Given our above analysis, we include the following tests: **ALG.1** is standard OLS for linear regression solved using normal equations [49]; **ALG.2** is standard logistic regression for binary classification solved using batch gradient descent [73]; **ALG.3** is non-negative matrix factorization for feature extraction solved using weighted multiplicative updates [52]; **ALG.4** is heterscedasticity-robust standard errors for OLS, which is common in the domain sciences and enterprises for handling heteroscedasticity [83]. Among these, **ALG.4** is *not* available as a native implementation in any of the compared systems (except MADlib). We now briefly discuss each algorithm and present their respective LA scripts.

Ordinary Least Squares (OLS). OLS is the most popular algorithm for linear regression [9, 49]. Given the data matrix $\mathbf{X}_{n\times d}$ and target vector $\mathbf{y}_{n\times 1}$ with values in $\mathbb{R}$, OLS computes a hyperplane $\mathbf{w}_{d\times 1}$ that minimizes the total squared Euclidean distance between $\mathbf{w}$ and the examples' target values. OLS offers clean interpretability of its co-efficients. It is also trivial to implement in LA systems, since it has a closed form analytical solution using the Gramian of $\mathbf{X}$ (if the Gramian is singular, iterative optimization algorithms are used). Algorithm 1 presents the LA script for OLS.

Algorithm 1: Ordinary Least Squares (OLS).

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

Logistic Regression (LR). LR is the most popular algorithm for classification [9]. Given $\mathbf{X}_{n\times d}$ and a class label vector $\mathbf{y}_{n\times 1}$ with labels in $\{0, 1\}$, LR computes a hyperplane $\mathbf{w}_{d\times 1}$ that optimizes a negative log-likelihood function that captures the log odds of the labels based on how far the data points are from $\mathbf{w}$. Iterative optimization algorithms are needed, since LR cannot be solved in closed form [73]. A simple algorithm is batch gradient descent (BGD); while it is seldom used directly for LR, its data access and communication behaviors are similar to more popular algorithms such as L-BFGS that typically converge in fewer iterations [73]. The convergence properties of optimization algorithms and ML accuracy are *orthogonal* to the LA system used; thus, we focus only on per-iteration runtimes. Algorithm 2 presents the LA script for LR with BGD. Note that SGD is also popular for LR on large data; but SGD is not amenable to a bulk LA-based implementation on the compared systems. Interestingly, MADlib, MLlib, and TensorFlow offer native LR implementations using SGD [31, 55, 71].

Algorithm 2: Logistic Regression (LR) with BGD.

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

Non-negative Matrix Factorization (NMF). NMF factorizes $\mathbf{X}_{n\times d}$ into two matrices $\mathbf{W}_{n\times r}$ and $\mathbf{H}_{r\times d}$ such that the reconstruction error $\|\mathbf{X} - \mathbf{WH}\|_F^2$ is minimized subject to all entries of $\mathbf{W}$ and $\mathbf{H}$ being non-negative. The user-specified parameter r is called the factorization rank. NMF is common for feature extraction, especially on sparse data with $d \gg r$ (e.g., for text mining), since it extracts r dense features. NMF can also use many optimization algorithms; weighted multiplicative updates is popular [33, 65]. Algorithm 3 presents its LA script. It stress tests matrix-matrix multiplication, including multiplication chains. Typical variants of NMF initialize $\mathbf{W}$ and $\mathbf{H}$ in a specific way to improve convergence behavior, e.g., Gaussian NMF uses Gaussian distributions [51]. As with LR, the convergence behavior of weighted multiplicative updates is orthogonal to the LA system and our focus, as are specific initialization criteria. Note that SGD can be used for NMF also [48].

Algorithm 3: NMF.

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

Heteroscedasticity-Robust Standard Errors (HRSE). Standard errors of OLS coefficients are used for formal hypothesis tests about the sign and magnitude of the coefficients in many domain sciences and enterprise settings. Heteroscedasticity is a key concern in this context. It arises if the variability of the target varies based on the feature values in linear regression. It makes conventional variance estimators for OLS inconsistent, which may lead to invalid hypothesis tests [84]. To mitigate this issue, "robust" heteroscedasticity aware variance estimators are popular, e.g., the procedure due to [83], shown in Algorithm 4. 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. Note that HRSE presents new opportunities for inter-operation optimization. For instance, while diag($\boldsymbol{\epsilon}$) is seemingly ultra-large ($n \times n$), it is just a diagonal matrix. Handling it as a vector will reduce the cost of the multiplication $\boldsymbol{X}^T\text{diag}(\boldsymbol{\epsilon})$. The potential for such optimizations by a scalable LA system makes HRSE an even more valuable test case.

Algorithm 4: HRSE.

```
Inputs: $\mathbf{X}$, $\boldsymbol{\epsilon}$: residuals from OLS
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 [80]. In particular, we focus primarily on the distributed memory setting here, which means most of our discussion centers on pBDR, MADlib, MLlib, and SystemML. Since TensorFlow is not (yet) natively distributed, we present most of its results, along with R and NumPy (the single-node baselines), in the technical report [80]. All the code 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 Setup. All experiments were run on CloudLab [75]. Each node runs Ubuntu 16.04 and has 200 GB of RAM, 24 CPUs, and 700 GB of disk. Versions of systems used are listed in the technical report [80]. All runs are repeated five times. The first runtime measurement is discarded (for warming the 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. Since the Spark-based systems (MLlib and SystemML) do not perform computations until the results are needed, we force the computation of specified operations by invoking a simple "count" on the result RDD. Furthermore, 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. 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 inevitable. But note that this writing cost is not needed for the other systems.

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 practices 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 [80]. Notwithstanding our tuning efforts, we do not regard the runtime results shown here as the final word on how fast each system can be. But our results shed new light on numerous performance trends and scalability bottlenecks of each system. 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 D.R. We fix C.N (number of nodes) at 8, D.C (d) at 100, at and vary D.R (n) for dense data. The data matrix is stored as a CSV file on disk, and varies in size from about 4 GB to 30 GB. For pbdR we generate random data in memory. For **MAT.6** in particular, we fix the number of columns of $\boldsymbol{N}$ at 100 and vary the number of rows, while the dimensions of $\boldsymbol{M}$ are $\boldsymbol{N}^T$. Figure 2 presents the results for four of the **MAT** tests (the other two are presented in the technical report [80]). We see that the runtime for each LA operation increases steadily, but the rate of increase is faster for **MAT.6**, which is consistent with their communication and computation behaviors explained in Section 3.1.1. All systems except MADlib finish in under 5s on **MAT.2** (norm) and **MAT.4** (matrix-vector multiplication). MADlib's LA operations are not as efficient as the others, likely due to higher per-tuple in-RDBMS processing overheads and increased IO time. MLlib's performance degrades dramatically on **MAT.5** and **MAT.6** compared to the other two tests. SystemML is almost always faster than both MLlib and MADlib, except on **MAT.5**. This is likely because SystemML simply pulls all data to the driver program and executes then in single-node mode in most cases. Overall, pBDR offers the best (or near-best) performance, which underscores both the overhead imposed by the other scalable LA systems and the importance of including such a strong baseline system when evaluating performance.

Multi-Node Sparse Data Vary D.S. We vary D.S for sparse data (fraction of nonzero entries in $\mathbf{X}$) on the same 8-node cluster. The dimensions are chosen such that $\boldsymbol{X}$ would be about 100 GB, if materialized as a CSV file. For **MAT.4**, $\mathbf{w}$ is dense. Note that we did not find support for sparse data in pBDR. Figure 3 presents results. These results are an interesting departure from that of dense data. While the overall trends are consistent with the communication and computation behaviors of the LA operations, the relative behaviors of the systems are markedly different. MADlib is now more competitive to SystemML, especially on **MAT.2** and **MAT.5**, wherein in runtimes increase proportionally to sparsity, since it stores one tuple per nonzero cell. MADlib reported an error on **MAT.4** because output is returned as an ARRAY rather than a table. At this data scale, this output array exceeded PostgreSQL's hard size limit of 1 GB for arrays. At the time of this writing, this is a serious scalability bottleneck for MADlib, since it

Figure 2: Multi-Node Dense Data for MAT with varying D.R.

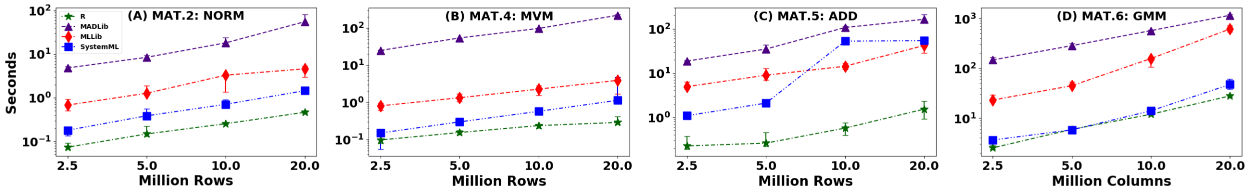


Figure 3: Multi-Node Sparse Data for MAT with varying D.S.

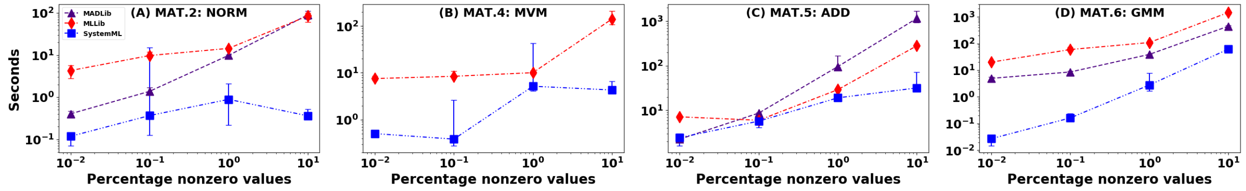


Figure 4: Single-Node Dense Data for MAT with varying C.C.

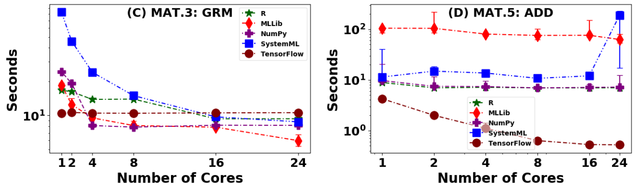
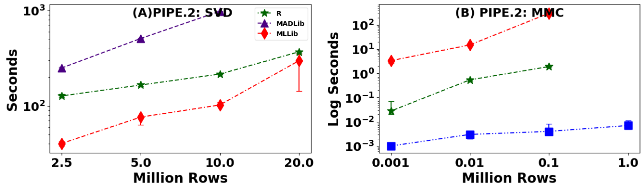


Figure 5: Multi-Node Dense Data for PIPE with varying D.R.



forces users to rewrite their LA scripts in a less intuitive manner to create table outputs instead of arrays. MLlib is also competitive on **MAT.5** but it is much slower on the other tests. Interestingly, MADlib outperforms MLlib on **MAT.6**, which can be explained by the efficiency of the RDBMS's join-aggregate optimizations. MLLib, however, densifies the right operand, which increases runtimes.

Single-Node Dense Data Vary C.C. We fix n at 20 million and d at 100. We vary C.C (number of cores) on a single node to study multicore speedup behaviors. Figure 4 presents the results for two key LA operations. We omit MADlib here, since it was much slower than the other systems and obfuscated the plot trends (MADlib's runtimes are in the technical report [80]). We see that the benefit of additional cores plateaus at between 4 and 16 cores depending on the system. In particular, for TensorFlow and R, more than 4 cores is not helpful for either **MAT.3** or **MAT.5**. While TensorFlow's runtimes are the lowest, we note that it uses single-precision floating points, while the other systems uses double-precision, which explains its roughly 2x gap with NumPy. SystemML, however, enjoys higher speedups even as C.C goes up to 24 on **MAT.3**, but it is significantly slower to start with at lower C.C.

4.2 Results for PIPE

Back to the 8-node setting, we now run **PIPE.1** (matrix chain multiplication) and **PIPE.2** (SVD) on dense data. For **PIPE.1**, we use the same chain from Section 3.1.2 and vary n. Figure 5(A) presents the results. As the plot shows, only SystemML determines the optimal multiplication order and avoids creating the large intermediate matrix. Both pbdR and MLlib have orders of magnitude higher runtimes and eventually crash for even a modestly large n. We note, however, that neither pbdR nor MLLib claim to be able to detect this optimization; thus, this is an avenue for improving these systems. MADlib is excluded from this test because users have to manually fix the matrix multiplication order when using its LA abstractions. Figure 5(B) presents the results for **PIPE.2**. SystemML is not shown because it does not provide distributed SVD. We fix d = 100 and vary n. We compute only the ten strongest singular values/vectors. Interestingly, MLlib's distributed SVD (which they describe in detail in [38]) is substantially more efficient than pbdR in most cases, which is in contrast to its slower performance on the **MAT** tests.

4.3 Results for ALG

Multi-Node Dense Data Vary D.R. For the same 8-node setup and data scales as Figure 2, we run the **ALG** tests (LA-based ML algorithms) for dense data. For LR and NMF, we set $I = 3$ as a more reliable proxy for per-iteration runtimes ($I = 1$). Figure 6 presents the results. Across the board, we see steadily increasing runtimes as the data scale increases, which is consistent with the **MAT** results, as well as the communication and computation behaviors discussed in Section 3.1.3. The only major difference with the relative trends seen in Figure 2 is how MLlib's performance is significantly worse here for all four algorithms. In fact, only SystemML exhibits performance that is competitive to the strong baseline pbdR on three tests. But even SystemML's performance degrades as n increases on **ALG.4** (HRSE), and it eventually crashes due to high metadata overheads for the ultra-sparse diagonal matrix in HRSE (see Algorithm 4). This issue did not arise in MADlib, MLlib, and pbdR because we could hand-code the multiplications in HRSE to avoid such metadata overheads (using R's "sweep" function

Figure 6: Multi-Node Dense Data for ALG with varying D.R.

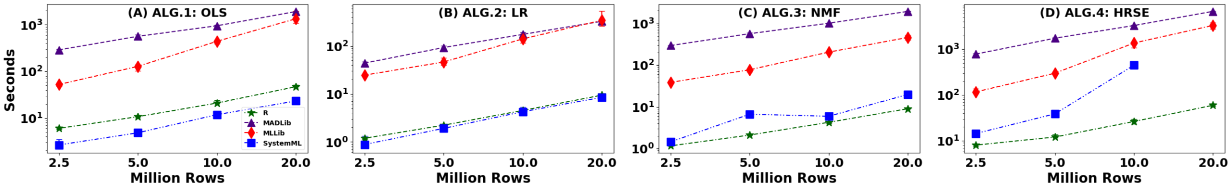


Figure 7: Single-Node Dense Data for ALG with varying C.C.

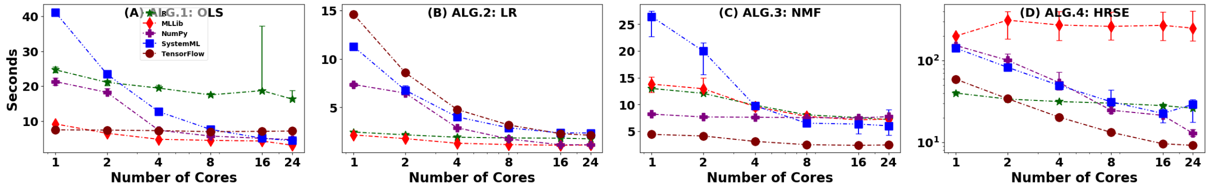


Table 3: LOC for Implementation of ALG Tests.

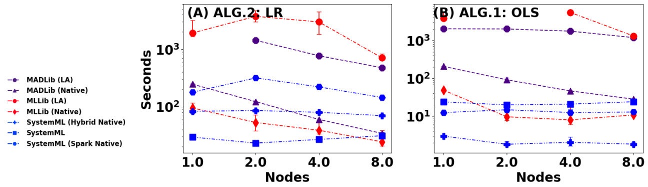
System	LR	OLS	NMF	HRSE
pbdR	12	3	9	5
SystemML	10	1	6	5
MADLib	27	12	46	19
MLLib	44	24	50	38
TensorFlow	23	12	19	12

for pbdR) at the cost of higher programming effort for users compared to SystemML's fully declarative approach. But MADlib and MLlib are still substantially slower on these **ALG** tests compared to pbdR (in all cases) and SystemML (in most cases). Interestingly, SystemML is slightly faster than pbdR on **ALG.1** (OLS).

Single-Node Dense Data Vary C.C. For the same single-node setup and data scales as Figure 4, we run the **ALG** tests for dense data to study the multicore speedup behaviors of all systems. For MADlib, we explcitily create new Greenplum instances with C.C segments. For Spark, we manually restrict the number of driver cores to C.C. Once again, we omit MADlib from this plot, since its runtimes are much higher and obscure the other trends (again, MADlib runtimes are in the technical report [80]). Figure 15 presents the results. The results are similar to the **MAT** results in the sense that most of these systems do not benefit from more than about 4–8 cores, except for SystemML, which has higher runtimes than TensorFlow and NumPy in most cases anyway. R is faster than NumPy on **ALG.2** and **ALG.4**, but not on the others, likely due to the overheads of intermediate data copies for matrix-matrix multiplication [79]. Surprisingly, MLlib is comparable in performance to TensorFlow on **ALG.1** and **ALG.2**. But it is slower on **ALG.3** and much slower on **ALG.4** due (we suspect) to the broadcasting method used to compute the dense-diagonal product in HRSE.

Implementation Effort. We now discuss briefly about the implementation effort required for writing the **ALG** tests on each system. Apart from the tuning effort explained earlier, this constitutes another key component of how easy or difficult such systems are to use. Ideally, one would need to conduct user studies with many real data scientists to get a thorough picture of system usability. In the absence of such extensive user studies, we think the lines of code (LOC) for expressing the LA-based algorithms tests serves as a reasonable surrogate for this comparison. Table 3 presents the relevant statistics.

Figure 8: Multi-Node *DenseCriteo* for LA-based and native implementations of ALG.1 and ALG.2.



Unsurprisingly, the systems that needed the lowest implementation efforts are SystemML and pbdR, since they have near-math syntax for expressing LA scripts. In contrast, MLlib needed the highest implementation efforts in terms of LOC because we often had to switch between different matrix datatypes using lower-level Scala code to ensure the whole script can run at scale. MADlib also needed rather high implementation effort, but with a mix of LA computations and SQL statements. Moreover, for the iterative algorithms, we had to write control programs in Python. TensorFlow was in between these two extremes with all computations expressed in its Python API. Overall, we found that pbdR and SystemML offered the best interface language that helps improve user productivity for expressing complex LA-based algorithms, while still executing on large-scale data transparently. Due to space constraints, we present more details about our LA-based algorithm implementations in the technical report [80], including code snippets for OLS in SystemML and TensorFlow. Note that all of our code for all the tests are available for download on our project webpage.

4.4 Scalability on Criteo Datasets

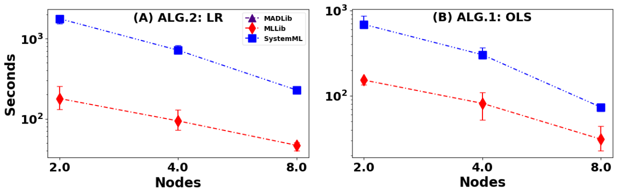
While our above results with synthetic data of various scales already give interesting insights about the compared systems, we also want to know how these systems perform on a large real-world dataset for ML analytics. In particular, we are also interested in comparing the native, non-LA based implementations offered by some systems against their own LA-based versions. We use a large public dataset from Criteo, whose full size is 1 TB [24]. 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 is about 32 GB (as a CSV file). 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 to sparse feature vectors based on one-hot encoding [13]. 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 repeatability.

We compare the scalability of the LA-based and native implementations of **ALG.1** (OLS) and **ALG.2** (LR) for MADlib, MLlib, and SystemML on our cluster by varying C.N (number of nodes). We skip **ALG.3** and **ALG.4** because most of these systems do not have native implementations of NMF and HRSE. 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 the IRLS optimization algorithm. MADlib's native LR offers a choice between conjugate gradient, IRLS, and SGD. We use SGD, since it has the lowest per-iteration runtime. MADlib's native OLS is solved directly as in 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 C.N. Figure 18 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 a new algorithm is really tedious to implement in lower-level code. Third, all systems exhibit near-linear speedups on LR (except LA-based LR in SystemML, which is the fastest), but almost none of them exhibit such speed-ups on OLS (except MLlib, which is the slowest). This sub-linearity suggests that communication costs for computing the Gramian in OLS remain a scalability bottleneck.

Figure 9: Multi-Node *SparseCriteo* for LA-based and native implementations of ALG.1 and ALG.2.



Multi-Node *SparseCriteo* Vary C.N. Figure 9 presents the results. This plot has much fewer lines than Figure 18 due to two reasons. First, the LA-based implementations of both MADlib nor MLlib, as well the native LR and OLS of MADlib were either incapable of running due to hard-coded scalability limits, or they crashed or timed out regardless of C.N! 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. Thus, we present the runtimes of SystemML's LA-based implementation in the technical report [80].

5. ANALYSIS AND DISCUSSION

We now discuss key lessons from our comparative evaluation, split into two parts: one aimed at practitioners interested in using scalable LA systems and one aimed at researchers interested in building/improving such systems.

5.1 Guidelines for Practitioners

We summarize each scalable LA system's strengths and weaknesses, organized as answers to the following practical questions. How difficult is the setup and 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? Are fault tolerance and automatic disk spills supported? How do LA-based ML implementations fare against native ML implementations? We skip discussing the single-node tools (R, NumPy, and TensorFlow), but note that a version of TensorFlow with distributed LA operations is under active development by the TensorFlow community [19].

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 via R's S4 multiple dispatch system; most R scripts can be combined with MPI 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 have to manually scale other algorithms. Overall, pbdR primarily targets users that write custom LA-based pipelines for self-contained data analysis.

MLlib. We found tuning MLlib to be complex, since Spark is sensitive to memory parameters. It often crashed with default settings. We performed best effort tuning, as explained before, but it is tedious. MLlib supports sparse matrix types, but it often densifies such data during execution, degrading performance. Physical data independence is limited, since users have to manually select the matrix type to use and decide partitioning and caching strategies. 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.

SystemML. We found tuning SystemML to be simpler than MLlib, since it abstracts away data layout and caching decisions. But in some cases, SystemML collected too large a result into its driver program memory and crashed, which required us to retune Spark's memory-related parameters. It 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. Since it has APIs to intermix DML and Spark queries, one can leverage Spark for data pre-processing and feature engineering. This allows users to retain the high degree of physical data independence offered by DML for scalable LA scripts. Since SystemML uses Spark RDDs, it inherits Spark's fault tolerance and disk spill capabilities. SystemML's native ML implementations 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.

MADlib. We found tuning MADlib on Greenplum to be relatively simple but laborious. The advice on setting the number of segments is not concrete [8], but as we explained before, we ran pre-tests to tune it. These pre-tests were laborious because changing the number of segments requires reloading the database. MADlib has generally good support for sparse data and was fast on the **MAT** tests with sparse data. But it has a scalability bottleneck due to the array size limit (1 GB) imposed by PostgreSQL/Greenplum. MADlib's use of an RDBMS gives it a reasonable level of physical data independence, since data partitioning and caching are abstracted away. But interleaving table creation in SQL and LA computations might be unintuitive for many users. 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; the RDBMS automates disk spills (except for the array type). MADlib's native ML implementations were typically faster than LA-based implementations.

5.2 Open Research Questions

Based on our extensive experiments with the state-of-the-art scalable LA systems, we identify a few key gaps that require more research from the data systems standpoint. We organize our discussion along three 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 real-world ML applications. The RDBMS-based MADlib faces high metadata and processing overhead for native join-aggregate query-based LA scripts, while MLlib and TensorFlow have relatively poor physical data independence. 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 clearly needed for fully supporting sparse data for efficient LA-based analytics at scale, including optimizing the data layout, staging computations and communication, and managing caching when executing entire LA scripts. This is challenging because we would need to predict the sparsity of intermediate matrices lest they get needlessly densified. This problem is akin to cardinality estimation for intermediate results in SQL query optimization. Adaptive optimization of LA scripts (e.g., with reinforcement learning) could be a helpful avenue for new research.

Auto-tuning Data Layout and System Parameters. The promise of physical data independence means users should not have to expend much efforts in 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 injecting caching directives, while MADlib required several data layout decisions for interleaving LA scripts with SQL table creation. Such data layout decisions could be unintuitive for statistical or ML-oriented users. MLlib and TensorFlow also require lower-level programming skills in Python (or Scala for MLlib) to handle 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-based analytics. Applying the lessons of auto-tuning relational and MapReduce workloads (e.g., [56, 81]) to this setting is another avenue for new research.

Multi-node "COST" and Parallelism Models. A surprising takeaway from our results relates to the bedrock of scalable data systems: data parallelism. Conventional wisdom is that using many cheap commodity nodes can reduce runtimes thanks to distributed memory, even at the scale of dozens of GBs, compared to using a smaller set of beefy but expensive nodes. But as our Criteo results show, increasing the number of nodes may not result in significant speedups - at least for the systems compared here. This is the multi-node counterpart of the well-known "COST" factor [69]. In fact, SystemML pulled all data into its driver program in many cases, since it deemed the single-node execution will be faster. It is not clear if this issue will get mitigated at larger data scales (e.g., TBs), since a larger data scale necessitates more nodes for larger total memory for the Spark-based systems to work well (MLlib and SystemML). In turn, this could increase communication costs for the underlying distributed LA operations, as well associated metadata and query processing overheads.

Furthermore, recent surveys show that about 70% of data scientists in practice analyze datasets no larger than 100 GB [10, 35]. Thus, while faster data-parallel LA implementations are useful, we think another fruitful research direction is to study transparent scalability for *task parallelism* across nodes with multi-core parallelism within a node along with disk spills. This can speed up crucial meta-level ML model selection tasks such as hyper-parameter tuning (HT) [35, 58, 60, 78]. There is some work on hybrid parallelism for SystemML on Hadoop, but it focused primarily on tasks over small subsets of the data rather than the whole dataset [37]. Similarly, a recent integration of TensorFlow with Spark enables users to perform HT for neural networks defined with TensorFlow but only for small datasets that can be broadcasted [4]. A related issue is determining an optimal cluster size for a given dataset and LA workload, similar to how [78] picks cluster sizes for a few specific ML algorithms. Devising an automated "optimizer" for deciding when to use data parallelism versus task parallelism, how to interweave the two parallelism models for complex LA scripts and/or native ML implementations with support for HT, while still offering high data independence, is a challenging but potentially impactful research direction.

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 highly nontrivial to standardize comparisons that consider runtimes and ML accuracy *simultaneously* precisely because the latter is closely tied to HT, which is typically a non-smooth and non-convex meta-level optimization problem [78]. Different systems will fall on different points of the accuracy-runtime *Pareto frontier* for different datasets, ML algorithms, and HT options, making an apples-to-apples comparison hard. Moreover, different implementations of the same ML task might have different hyper-parameters to tune. For instance, how does one standardize how HT should be done when comparing, say, MADlib's SGD-based LR with SystemML's native LR? Should sub-sampling be allowed for faster HT? Should more time be spent on larger hyper-parameter grids? We leave such questions to future work, but note that there is growing interest in standardizing such Pareto frontier-based evaluations, at least for deep learning tasks [17].

6. OTHER RELATED WORK

Benchmarks of LA Packages. LA packages such as BLAS, LAPACK, ScaLAPACK, and Eigen have been extensively benchmarked [32, 34, 45, 46, 63, 72]. For instance, the LINPACK benchmark focuses on the efficiency (measured in MFLOPs per second) of solving a system of linear equations [44, 62]. 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 [62] 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 [47, 61]. 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 [79].

Comparisons of Analytics Systems. Since MLlib, SystemML, MADlib, and TensorFlow were released in their current form only within the last three–four years, there is no known comparative evaluation of their performance for scalable LA workloads. While their reference publications show several results [31, 36, 55, 71], there is a lack of uniformity in the workloads, data scales, and computational environments studied. Our work serves to fill this crucial gap in the literature by comparing them on an even footing. Recent work has also evaluated the performance of Apache Mahout Samsara, but only for a few operations [76]. [70] compared SciDB [39], Myria [82], Spark, and TensorFlow for a specific scientific image processing task but not for general LA or ML workloads. [41] compare Spark, GraphLab [66], SimSQL [40], and Giraph [2] for Bayesian ML models implemented using the lower-level abstractions of such systems; they do not evaluate LA operations LA-based ML algorithms. [35] compared MLlib's native ML implementations with efficient single-node ML tools such as Vowpal Wabbit [21] 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 operations and LA-based data analysis workloads, not just a few specific ML algorithms. We also include MADlib, SystemML, and the oft-ignored pbdR/ScaLAPACK for the comparisons. Finally, [17] introduce a 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 cover bulk LA workloads or structured data analytics. Thus, overall, all these prior benchmarking efforts are largely orthogonal to our work.

7. REFERENCES

[1] Amazon Web Services ML. https://aws.amazon.com/machine-learning/.

[2] Apache giraph. http://giraph.apache.org.

[3] Apache SystemML Webpage. https://systemml.apache.org/.

[4] Deep learning with apache spark and tensorflow. https://databricks.com/blog/2016/01/25/deep-learning-with-apache-spark-and-tensorflow.html.

[5] Distributed TensorFlow. https://www.tensorflow.org/deploy/distributed.

[6] Gartner Report on Analytics. gartner.com/it/page.jsp?id=1971516.

[7] Google Cloud ML Engine. https://cloud.google.com/ml-engine/.

[8] Greenplum Tuning Guidelines for Number of Segments. https://gpdb.docs.pivotal.io/530/admin_guide/intro/arch_overview.html#arch_segments.

[9] Kaggle survey: The state of data science and ml. https://www.kaggle.com/surveys/2017. Accessed January 31, 2018.

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

[11] Microsoft Azure ML. https://azure.microsoft.com/en-us/services/machine-learning-studio/.

[12] Microsoft Revolution R. http://blog.revolutionanalytics.com/2016/01/microsoft-r-open.html.

[13] One-Hot Encoding Example in Scikit-learn. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[14] Oracle R Enterprise. www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html.

[15] Project R. r-project.org.

[16] SparkR. spark.apache.org/R.

[17] Stanford dawnbench: An end-to-end deep learning benchmark and competition. https://dawn.cs.stanford.edu/benchmark/.

[18] TensorFlow Accelerated Linear Algebra (XLA). https://www.tensorflow.org/performance/xla/.

[19] TensorFlow Webpage. https://www.tensorflow.org/.

[20] The Comprehensive R Archive Networks. https://cran.r-project.org/.

[21] Vowpal wabbit. https://github.com/JohnLangford/vowpal_wabbit/wiki.

[22] 2017 big data analytics market survey summary, 2017. https://www.forbes.com/sites/louiscolumbus/2017/12/24/53-of-companies-are-adopting-big-data-analytics/#4b513fce39a1.

[23] Cloudlab user manual, 2017. http://docs.cloudlab.us/hardware.html.

[24] Criteo terrabyte click logs, 2017. http://labs.criteo.com/2013/12/download-terabyte-click-logs/.

[25] Install greenplum oss on ubuntu, 2017. https://greenplum.org/install-greenplum-oss-on-ubuntu/.

[26] Installing tensorflow from source, 2017. https://www.tensorflow.org/install/install_sources.

[27] Openstack user manual, 2017. https://docs.openstack.org/pike/.

[28] Spark mllib data types - rdd based api, 2017. https://spark.apache.org/docs/latest/mllib-data-types.html#data-types-rdd-based-api.

[29] Spark openstack, 2017. https://github.com/ispras/spark-openstack.

[30] Using native blas in systemml, 2017. https://apache.github.io/systemml/native-backend.

[31] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[32] 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.

[33] 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.

[34] 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.

[35] C. Boden, T. Rabl, and V. Markl. Distributed Machine Learning - but at what COST? In *NIPS MLSys Workshop*, 2017.

[36] 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.

[37] 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. *Proc. VLDB Endow.*, 7(7):553–564, Mar. 2014.

[38] R. Bosagh Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. Sparks, A. Staple, and M. Zaharia. Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38. ACM, 2016.

[39] 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*, pages 963–968. ACM, 2010.

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

[41] 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*, pages 1371–1382. ACM, 2014.

[42] L. Chen, A. Kumar, J. Naughton, and J. M. Patel. Towards linear algebra over normalized data. *Proc. VLDB Endow.*, 10(11):1214–1225, Aug. 2017. `https://doi.org/10.14778/3137628.3137633`.

[43] 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.

[44] J. Dongarra et al. *LINPACK Users' Guide*. 1979.

[45] J. Dongarra et al. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[46] J. Dongarra et al. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.

[47] 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.

[48] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD*, 2012.

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

[50] 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, Budapest, Hungary, September 2004.

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

[52] N. Gillis. Introduction to nonnegative matrix factorization. *arXiv preprint arXiv:1703.00663*, 2017.

[53] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[54] D. Hall and D. Ramage. *Breeze Documentation*, 2016. `https://github.com/scalanlp/breeze/wiki`.

[55] 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.

[56] 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 *In CIDR*, pages 261–272, 2011.

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc 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, 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, and J. Ross. In-datacenter performance analysis of a tensor processing unit. 2017. `https://arxiv.org/pdf/1704.04760.pdf`.

[58] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[59] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1717–1722. ACM, 2017.

[60] 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.

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

[62] J. Kurzak, J. Dongarra, M. Heroux, and J. Demmel. Linear Algebra Libraries for High-Performance Computing: Scientific Computing with Multicore and Accelerators. In *SC*, 2017.

[63] C. L. Lawson et al. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[64] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[65] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788, 1999.

[66] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*, 2010.

[67] 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 (ICDE)*, pages 523–534, April 2017.

[68] MADLib development team. Madlib user documentation, 2017. `http://madlib.incubator.apache.org/docs/latest/index.html`.

[69] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[70] 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. *arXiv preprint arXiv:1612.02485*, 2016.

[71] 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.

[72] MKL Development Team. *Intel Math Kernel Library Developer Reference*. Intel Corporation, 2015. `https://software.intel.com/en-us/articles/mkl-reference-manual`.

[73] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 2006.

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

[75] R. Ricci, E. Eide, and the CloudLab Team. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *;login:*, 39(6), 2014.

[76] 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 NIPS*, 2016.

[77] D. Schmidt, W.-C. Chen, G. Ostrouchov, and P. Patel. *A Quick Guide for the pbdDMAT Package*. R package vignette.

[78] 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.

[79] S. Sridharan and J. M. Patel. Profiling r on a contemporary processor. *Proceedings of the VLDB Endowment*, 8(2):173–184, 2014.

[80] 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`.

[81] 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 International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, 2017. ACM. `http://doi.acm.org/10.1145/3035918.3064029`.

[82] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Hayes, B. Howe, D. Hutchinson, S. Jain, R. Maas, P. Mehta, et al. The myria big data management and analytics system and cloud services, 2017. CIDR.

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

[84] J. M. Wooldridge. *Introductory econometrics: A modern approach*. Nelson Education, 2015.

[85] Y. Zhang, H. Herodotou, and J. Yang. Riot: I/o-efficient numerical computing without sql. *arXiv preprint arXiv:0909.1766*, 2009.

[86] Y. Zhang, W. Zhang, and J. Yang. I/O-Efficient Statistical Computing with RIOT. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, 2010.

8. APPENDIX

8.1 Additional Background on Systems Compared

Table 4 presents a summary of salient characteristics of each system in the slate we compare. We additionally provide example implementations of OLS in each of these systems. Note that LOC counts may not agree exactly with the numbers reported in table 3 in the body. The example programs here have been reformatted to display cleanly and exclude some "utility" routines which are needed to make them work in practice. The following sections discuss these examples.

Numpy, R, and SystemML. Program 2 shows the implementation in NumPy. Compare this script to program 4 which presents the corresponding algorithm in R/pbdR. The `init.grid()` and `finalize()` statements are used by pbdR to initialize and finalize the MPI communicator. The pure R implementation is identical less these statements. These scripts also highlight the differing paradigms between R and NumPy. NumPy favors an object oriented syntax while R's is imperative. As can be seen in program 1, SystemML's syntax is virtually identical to pbdR (and yields comparable performance!). SystemML also allows users to pass metadata about input matrices which can help the optimizer determine better execution plans.

TensorFlow. Program 5 presents OLS in TensorFlow's low level LA API. While TF bears some superficial similarity to NumPy, its syntax and programming model are quire different. The lines of code between the graph declaration and the `tf.Session()` constructor merely place operations on the computation graph. When `run()` is called on an object from the graph, TensorFlow compiles the graph into an execution plan and runs the actual computation.

MADLib. Program 6 presents an implementation of OLS using MADLib's matrix API. Note that it is not possible to pipeline operations in MADLib as in the other languages. The MADLib API requires that each intermediate stage of computation be explicitly materialized. We remark that although this code explicitly computes the inverse of the Gram matrix, the number of columns we use is small and so this step remains low cost.

MLLib. Program 3 presents an implementation of OLS using the distributed matrix types provided by MLLib. The script highlights two of the distributed types provided by MLLib. We store the input matrix $\boldsymbol{X}$ as a *IndexedRowMatrix* which represents a distributed matrix using an RDD of local vectors. We selected this datatype because it provides an extremely efficient routine for computing the Gram matrix. However, it supports neither transposition nor multiplication with another distributed matrix and so we must cast it to a *BlockMatrix* to transpose and multiply with $\boldsymbol{y}$. This highlights several important implementation decisions which pbdR and SystemML abstract from the user. It is unclear whether it would be more efficient to simply store everything as a *BlockMatrix* to begin with and eliminate the step of casting $\boldsymbol{X}$. However, then we forgo the optimized routine to compute a Gram matrix provided by *IndexedRowMatrix* and must make do with the (much) slower general matrix-matrix multiplication method provided by *BlockMatrix*. Additionally, we could have stored $\boldsymbol{y}$ as an *LocalMatrix* and used the "multiply" method of *IndexedRowMatrix* which supports multiplication with a local matrix. However, then our code would break if $\boldsymbol{y}$ exceeds single node memory. Even then, we would have to cast $\boldsymbol{X}$ to a *CoordinateMatrix* and then back to an *IndexedRowMatrix* to perform the operation which may result in shuffle. Without testing it is unclear which approach is better. Finally, we note that it is necessary to select the block sizes used for the *BlockMatrix*. Naively accepting the default of 1024 × 1024 would be catastrophic as the blocks are created as sparse matrices, but are promoted to dense during multiplication. This would then result in significant wasted space. This lack of strong physical data independence leads to some implementation headaches with MLLib.

8.2 Additional Detail on Tuning and Configuration

In the following section we describe the process used to tune each system and provide results from a series of "mini-tests" designed to test configuration parameters.

Choice of Linear Algebra Library. Both SystemML and MLLib use a JVM-based linear algebra library (Apache Commons Math and Netlib-Java respectively) by default, but allow users to import a system optimized BLAS library at runtime. Native BLAS is well known to outperform JVM-based implementations and so we first consider the effect of LA library on SystemML and MLLib performance. We compile the popular OpenBLAS library from source following the instructions in [30]. Table 10 shows the effect of using the JVM-based BLAS vs. OpenBLAS for GMM in the single node setting. We find only a small and inconsistent effect of using OpenBLAS with SystemML and so retain the default LA library in all SystemML tests as this makes implementation simpler. Tables 11 and 12 show the effect of using OpenBLAS in the single node and distributed settings for Spark MLLib. We find a strong positive effect of using OpenBLAS in the single node setting, but only small and inconsistent effects in the distributed setting. For GMM this small benefit is likely because the *BlockMatrix* constructor recommended by the documentation construct *sparse* matrix blocks [28]. Sparse matrix computations are not optimized by the native BLAS. Based on these tests we use OpenBLAS for single node MLLib tests but retain the JVM implementation in the distributed setting.

Additional Tuning Consideration for Spark MLLib. An additional consideration with Spark is the number partitions in RDDs underlying its matrix types. Ad-hoc comparison indicated that between 500 and 1000 partitions yielded good performance. Another important consideration is the amount of RAM and CPUs allocated to each executor. When using the standalone cluster manager, Spark calculates the number of executor instances based on these parameter settings. We perform an ad-hoc tuning of these parameter settings for matrix multiplication. Users are sometimes advised to use several executors with fewer cores and smaller amounts of RAM over "fat" executors which have all cores and RAM available. We find this indeed led to modest speedups ($< 2x$) for multiplication of smaller matrices but led to errors on the largest datasets we considered. Because

Table 4: Key characteristics of systems compared. "Applicable Environments" are the environments the system was primarily designed for: "SM" is single-node in-memory, "SD" is single-node disk-capable, "DM" is distributed memory, and "DD" is distributed disk-capable. By "Partial Declarativity," we mean that the system optimizes LA scripts only in a limited way or not at all even if it supports alternative physical implementations of LA operations. "Full Declarativity" means the system optimizes LA scripts both logically and physically. Base R does not support sparse matrices but user packages fill this gap. TensorFlow has a sparse matrix library but it has limited support for sparse LA operations.

	R ; NumPy	**MADlib**	**TensorFlow**	**SystemML**	**MLlib**
Applicable Environment(s)	SM	SD, DD	SM, DM	DM, DD	DM, DD
Interface Language(s)	R ; Python	SQL	Python, `C++`	DM, Python	Python, Scala
Storage Back-End	In-memory	RDBMS	Flat files	Spark/HDFS	Spark/HDFS
Declarativity	None	Partial	Partial	Full	Partial
Optimization	None	Cost-Based	Cost-Based	Cost-Based	None
Sparse Matrix Support	Yes	Yes	Partial	Yes	Partial
Implementation Language	C	`C++/SQL`	`C++`	Java/Scala	Scala
Linear Algebra Library	BLAS	Eigen	Eigen	Apache Commons Math	Breeze (JBLAS)

the speedups seemed to be small and this led to problems at scale, we adopted the "fat executor" policy and use a single executor per node (the master node runs both the driver and an executor). We perform a tuning mini-test to determine the optimal number of cores to allocate to the executor. With too many cores, there is a risk that the overhead of parallelism may begin to exceed the benefits. Table 13 presents results from a test which varies the number of executor cores for our core matrix operators on a four node cluster. We find that allocating all twenty four cores to the executor yields optimal performance.

In yet another tunable setting, MLLib allows users to adjust the number of "mid-dim-splits" used during distributed matrix multiplication. Tuning this parameter can lead to increased parallelism and reduced shuffle. Table 14 compares various settings of this parameter. We find 500 mid-dim-splits to be optimal and so use this number for all distributed matrix multiplications in MLLib.

Greenplum. A key tuning consideration for Greenplum is the number of segments used per cluster node. Each segment corresponds to a Postgres database instance which communicates with other segments to perform work. Too few segments may result in low parallelism while too many may lead to excessive communication overhead. Greenplum documentation states that common practice is to use between two and eight segments per host. Table 16 compares the effect of using six, twelve and twenty-four segments per host on a cluster with eight nodes. We find that six segments per node yielded the best performance and so use this setting for all distributed tests. In the single node setting we found that sixteen to twenty four segments yielded good performance and so use twenty four segments in the single node setting. Greenplum also allows users to tune the memory available to the DBMS. We set `gp_vmem_protect_limit=13000` which allows each Postgres instance to use up to 13000MB of RAM. We note that we saw only very little difference between modifying this setting and leaving it at its default. Greenplum additionally allows users to configure memory for individual queries through use of the `SET STATEMENT_MEM = 'X'` command. We tried various values of statement memory but found either little effect or that they resulted in memory errors. We therefore leave `STATEMENT_MEM` at its default value (2000MB).

Program 1: OLS in SystemML

```
reg = function(matrix[double] X,
               matrix[double] y)
    return (matrix[double] b) {
    b = solve(t(X) %*% X, t(X) %*% y)
}
```

Program 2: OLS in NumPy

```
def reg(X,y):
    return alg.solve(X.T.dot(X), X.T.dot(y))
```

8.3 Test Environment and Configuration

In the following section we discuss relevant aspects of the environment used to perform tests. All tests were performed on the CloudLab [75] "Clemson" site using `c6320` instance types. We are grateful to CloudLab for providing the infrastructure for this project. Table 6, reproduced from [23], describes the characteristics of the physical nodes used to perform experiments. On top of these physical nodes we use OpenStack [27] to create and manage a virtual cluster. Scripts to create and manage clusters (adapted from [29]) are available on the project github page. Each virtual node in OpenStack is provisioned as described in table 7. Table 8 describes the version of relevant software used for testing. The following sections discuss relevant installation and configuration details for each software package used. We stress that it is not necessary to manually replicate these steps. Scripts are available which automate configuration of cluster nodes.

Spark. In the single node setting we compile Spark from source using the command `mvn -DskipTests -Pnetlib-lgpl clean package` in order to take advantage of native BLAS accelerations. In the distributed setting we use a precompiled binary downloaded from Apache. We write all Spark code using the Scala API and create fat JAR files using SBT assembly. We run Spark using the standalone clus-

Program 3: OLS in MLLib

```
def reg(X: IndexedRowMatrix,
        y: IndexedRowMatrix) : Matrix = {
  val XTX = X.computeGramianMatrix()
  val XTY = X.toBlockMatrix(
                1024,X.numCols.toInt).
                transpose.multiply(
                    y.toBlockMatrix(1024,1), 500
                ).toLocalMatrix
  val b = from_breeze(
    to_dense(as_breeze(XTX)) \
    to_dense(as_breeze(XTY)))
  return b
}
```

Program 4: OLS in pbdR

```
library(pbdDMAT)
init.grid()

reg <- function(X, y) {
    b <- solve(t(X) %*% X, t(X) %*% y)
    return(b)
}

finalize()
```

Program 5: OLS in TensorFlow's LA API

```
def reg(Xdata, ydata):
  G = tf.Graph()
  with G.as_default():
    X = tf.placeholder(tf.float32,
          shape=Xdata.shape)
    y = tf.placeholder(tf.float32,
          shape=ydata.shape)

    b = tf.matrix_solve(
          tf.matmul(X, X, transpose_a=True),
          tf.matmul(X, y, transpose_a=True)
        )

    init = tf.global_variables_initializer()
    with tf.Session() as sess:
      sess.run(init)
      res = sess.run(b,
        feed_dict={X: Xdata, y: ydata})

    return res
```

Program 6: OLS in MADLib

```
  DROP TABLE IF EXISTS XT
  DROP TABLE IF EXISTS XTX
  DROP TABLE IF EXISTS XTY
  DROP TABLE IF EXISTS XTX_INV
  DROP TABLE IF EXISTS B

  SELECT madlib.matrix_mult(
     'X', 'trans=True', 'X',NULL, 'XTX');
  SELECT madlib.matrix_mult(
     'X', 'trans=True', 'y',NULL, 'XTY');
  SELECT madlib.matrix_inverse(
     'XTX',NULL, 'XTX_INV',NULL);
  SELECT madlib.matrix_mult(
     'XTX_INV',NULL, 'XTY',NULL, 'B');
```

ter manager (as opposed to YARN) and submit JARs using `spark-submit`. In the single node setting we configure Spark to import OpenBLAS as the LA backend at runtime. Instructions to do this are specific to the OS and LA library. The node configuration script provided in the project repo provides instructions for Ubuntu 16.04.

OpenBLAS. In the single node setting we compile OpenBLAS from source as described in [30] and use OpenMP as the threading implementation. We pull the OpenBLAS repo at revision `114fc0bae3a` and compile using `sudo make USE_OPENMP=1`. We found it was necessary to manually add some symbolic links to coax SystemML and MLLib into using this BLAS.

Greenplum and MADLib. We compile Greenplum from source as described in the cluster configuration script available on the project github page. We note that since the project began, Pivotal has made a PPA for Ubuntu available which allows Greenplum to be easily installed using `apt-get` [25]. Users may wish to try this method first as compiling Greenplum from source is a non-trivial process. Following the advice available on the Greenplum github page we configure several system parameters as described in table 5. We compiled MADLib from source (see `install-madlib.sh`). We note that it is important to use GCC/G++ 4.9 which must be installed separately from the Ubuntu repositories. Using the default GCC 5.x resulted in errors at runtime.

TensorFlow. We compile TensorFlow from source as described in [26]. We do not enable any of the extra packages available during the "configure" stage of installation and do not build with GPU support.

SystemML. We compile SystemML from source using revision `d91d24a9fa`. We built from this version because it contained patches designed to address an issue we encountered using sparse matrices. We compiled using `mvn clean package` and then manually copied the resulting `.jar` file to the `/lib` folder of each SBT assembly directory. SBT will then automatically package the JAR with other code. The precompiled JAR can be obtained by simply downloading the relevant directories from the project github.

pbdR. We install pbdR using R's built in package manager. R expects to link to a BLAS library at runtime. In the distributed setting we install OpenBLAS using `apt-get` on each node. Because of the "SPMD" programming model used by MPI, each R process must have access to the source file containing code to be executed. To ensure R processes in remote nodes have access to both the test script and the pbdR library source, we NFS share the home directory over the cluster. We use the "OpenMPI" [50] installation available through the Ubuntu repositories.

Misc. Single Node Tools. We install R in the single node setting from the "R-Studio" PPA using `apt-get`. We configured R to link against the version of OpenBLAS compiled from source (in the single node setting only). We installed NumPy using `pip install`. NumPy ships with a built in BLAS implementation which performs quite well and so we did not link against the native library.

8.4 Additional Experimental Results

We here present and discuss additional results not presented in the body.

8.4.1 Additional Results for MAT

Multi-Node Dense Data Vary D.R. Figure 10 presents the remaining matrix operators for tests which fix the cluster size at eight nodes and vary the number of rows in input matrices. We point out that Spark MLLib has an optimized Gram matrix computation routine which performs very well - even beating out pbdR which typically led the pack for distributed matrix ops. We make one concluding remark about matrix operator tests for MLLib. For **MAT.5** (matrix addition), only the *BlockMatrix* type provides a method supporting addition with another distributed matrix. We found this method to be quite slow in practice. Because of this, we implement a simple "add" function for the indexed row type which joins a pair of row matrices by row id and then maps over the resulting pair RDD, producing a new RDD which is the sum of the row vectors. We present a comparison in table 15. This method substantially out-performed the built in method and so all numbers reported in plots use this approach.

Multi-Node Sparse Data Vary D.S. Figure 11 presents the remaining matrix operators for tests which fix the cluster size at eight nodes and vary the sparsity of input matrices. We remark that MADLib crashed during computation of Gram matrix for the largest matrix size due to an attempt to materialize a large array. We note that SystemML outperforms MLLib's optimized Gram matrix computation on sparse data, although this is likely because it is performing computation in the driver.

Multi-Node Dense Data Vary C.N. Figure 12 presents results from tests which fix matrix dimensions at 20 million rows by 100 columns and vary the number of nodes in the cluster from two to eight. We note that speedups from scaling the number of nodes are remarkably small - especially for MADLib. We note as well that MLLib timed out for GMM in the two node setting.

Multi-Node Sparse Data Vary C.N. Figure 13 presents results from tests which fix data sparsity at 1% and vary the number of nodes in the cluster from two to eight. The upper panel plots raw runtime, and the bottom plots speedup curves relative to a single core. MADLib timed out for GRM computation on a single segment. SystemML shows little consistent benefit from scaling the number of nodes. This is because it is pulling data into the driver and performing computation in single node mode. Interestingly, MLLib and MADLib show slightly more benefit from adding cluster nodes in the sparse setting than dense.

Single-Node Dense Data Vary D.R. Figure 14 presents results from tests which scale the number of rows in input matrices in the single node setting. Matrix sizes are as in distributed tests. Note that we here introduce new baseline systems - R and NumPy. In this context, R is conventional single node R as opposed to the pbdR flavor used in the distributed setting. In the single node setting we use MLLib's *local* matrix types. In the local setting we found that 24 segments yielded optimal (or close to optimal) performance for Greenplum and so all numbers reported here use 24 segments. With the exception of MADLib, performance of all systems is fairly consistent. This is unsurprising as all systems are calling out to libraries which have been heavily optimized for the single-node in memory setting. R suffers from some well known "copy overhead" in certain contexts [79] which likely explains its modest gap relative to other systems on norm and MVM. We note that using a native BLAS was critical to obtaining good performance from MLLib in the single node setting. Using the JVM-based implementation bundled with Spark resulted in poor performance relative to other systems.

Single-Node Dense Data Vary C.C. Figure 16 presents results from tests which scale the number of CPU cores for selected single node dense matrix operators. The top panel plots raw numbers (including MADLib) while the bottom plots a "speedup curve" relative to time with a single core. As noted previously, we use the unix `taskset` command to pin each process to a specific subset of CPU cores. For Spark-based systems we explicitly restrict the number of driver cores. For Greenplum, we build separate database instances with the stipulated number of segments. As was remarked previously, speedups (at least on the operators considered) are generally sublinear. Interestingly, several systems exhibit different speedup behaviors between the two operators. This likely reflects differing parallelization strategies of the underlying linear algebra library used by each system. For example, Eigen multi-threads only a small subset of the operators parallelized by OpenBLAS.

8.4.2 Additional Results for ALG

Single-Node Dense Data Vary C.C. Figure 15 presents complete results from tests which fix the size of input data and vary the number of cores allocated. For all systems but MADLib we fix the size of input data at $10,000,000 \times 100$. For MADLib, we fix the size of data at $2,500,000 \times 100$ to obtain tractable run-times. The top panel plots raw run-times while the bottom plots "speedup curves" relative to the run-time for a single core.

Multi-Node Sparse Data Vary D.S. Figure 17 presents results from tests which fix the cluster size at eight nodes and vary data sparsity. We here present results only for MLLib and SystemML as MADLib either timed out or crashed

Table 5: System Parameters Used

Parameter Name	Value
/etc/security/limits.conf	
nofile (soft/hard)	131093
nproc (soft/hard)	131072
/etc/security/limits.conf	
net.ipv6.conf.all.disable_ipv6	1
net.ipv4.tcp_syncookies	0
net.ipv4.conf.default.accept_source_route	0
net.ipv4.tcp_tw_recycle	1
net.ipv4.tcp_max_syn_backlog	4096
net.ipv4.conf.all.arp_filter	1
net.ipv4.ip_local_port_range	1025 65535
net.core.netdev_max_backlog	10000
net.core.rmem_max	2097152
net.core.wmem_max	2097152
vm.overcommit_memory	1
kernel.shmmax	500000000
kernel.shmmni	4096
kernel.shmall	4000000000
kernel.sem	250 512000 100 2048
net.ipv6.conf.lo.disable_ipv6	1

Table 6: CloudLab Node Hardware

CPU	Two Intel E5-2683 v3 14-core CPUs (2.00 GHz)
RAM	256GB ECC Memory
Disk	Two 1 TB 7.2K RPM 3G SATA HDDs
Network	Dual-port Intel 10Gbe NIC (X520)

Table 7: OpenStack Instance Traits

CPU	24
RAM	204000MB
Disk	700GB
OS	Ubuntu 16.04

when attempting to allocate a large array for both tests. We present results only for GNMF and HRSE as LR and REG were examined in tests on *SparseCriteo*. Somewhat surprisingly, we found that SystemML executed substantially faster when forced to run in "spark-only" mode. Using our default driver memory of 32G, SystemML attempted to perform the computation in the driver and crashed. We had to increase driver memory to 80G before SystemML could complete these tests. However, we found that *reducing* driver memory to 10G, which forces SystemML to perform most computation on Spark, resulted in roughly 3x better performance. By default, SystemML will execute computation in the driver if it estimates driver memory is sufficient. The results of this test highlight two important issues - first, even if computation *can* fit in the driver, it may not be optimal to do so, and second, SystemML often does not correctly estimate the memory footprint of computations which necessitates tedious trial and error tuning of JVM heap sizes.

Multi-Node *DenseCriteo* Vary C.N. Figure 18 presents results from tests which compare native and LA-based PCA implementation on the dense Criteo AdClick dataset. We compute the five strongest principal components. All tests additionally reproject the input data $\boldsymbol{X}$ under the corresponding principal components. For our LA-based implementation, we compute the principal components using an eigen decomposition of the data covariance matrix.

Multi-Node *SparseCriteo* Vary C.N. Table 9 presents results from our LA-based implementations of logistic regression run on the sparse Criteo dataset. We report only logistic regression because this was the only test which could be completed by any system. Both PCA and OLS regression crashed when attempting to materialize the large $\boldsymbol{X}^T\boldsymbol{X}$ matrix necessary for both algorithms. The table contains results only for SystemML as we could not obtain results for any other system due to timeouts.

Table 8: Software Packages and Versions

Name	Version	How Installed?
Spark (Distributed)	2.2.0	Precompiled binary
Spark (Single Node)	2.2.0	Compile from source
Hadoop	2.7.3	Precompiled binary
R	3.4.3	Precompiled binary
Python	2.7.12	Precompiled binary
TensorFlow	1.4.1	Compile from source
NumPy	1.14.0	Precompiled binary
OpenBLAS (Single Node)	`114fc0bae3a`	Compile from source
OpenBLAS (Distributed)	Precompiled binary	
SystemML	`d91d24a9fa`	Compile from source
MADLib	1.12	Compile from source
Greenplum	5.1.0	Compile from source

Table 9: Multi-Node *SparseCriteo* for SystemML with varying D.N.

Nodes	Run Times (Seconds) - LR				
2	13,870.24	7,367.51	6,972.21	7,643.25	7,033.11
4	3,498.50	2,004.59	2,059.89	2,399.09	2,471.37
8	505.08	450.20	432.31	428.14	445.43

Table 10: Effect of LA Library on SystemML Performance (Single Node)

	GMM Run Time (Seconds)				
Open BLAS	10.58	11.93	10.75	11.44	11.17
Commons Math	7.96	8.49	10.14	13.60	14.45

Table 11: Effect of LA Library on MLLib Performance (8 Nodes)

	GMM Run Time (Seconds)				
Open BLAS	3,404.50	1,872.21	2,068.55	1,587.44	1,727.96
Netlib-Java	2,597.51	1,454.14	1,378.21	1,859.64	1,537.38
	TSM Run Time (Seconds)				
Open BLAS	38.71	29.31	25.37	25.27	23.88
Netlib-Java	5.91	2.11	2.07	3.73	3.32
	ADD Run Time (Seconds)				
Open BLAS	77.67	46.97	37.80	29.50	34.36
Netlib-Java	159.37	38.30	46.11	47.27	28.24
	NORM Run Time (Seconds)				
Open BLAS	3.02	4.55	2.34	4.61	3.49
Netlib-Java	5.33	4.55	4.48	4.54	4.47
	MVM Run Time (Seconds)				
Open BLAS	5.45	1.84	2.86	2.40	1.67
Netlib-Java	5.25	3.60	2.57	3.36	2.57

Table 12: Effect of LA Library on MLLib Performance (Single Node)

	GMM Run Time (Seconds)				
Open BLAS	64.84	47.56	7.82	6.68	6.50
Netlib-Java	282.14	280.85	284.02	289.75	276.72
	ADD Run Time (Seconds)				
Open BLAS	223.88	195.60	58.82	85.26	58.76
Netlib-Java	258.83	79.00	64.46	53.96	60.97
	NORM Run Time (Seconds)				
Open BLAS	144.17	312.43	69.13	103.51	65.35
Netlib-Java	86.40	211.62	68.58	124.10	69.49
	MVM Run Time (Seconds)				
Open BLAS	0.72	0.36	0.39	1.45	0.36
Netlib-Java	3.92	3.66	3.75	5.30	3.32

Table 13: Effect of Number of Executor Cores on MLLib Performance (4 Nodes)

	GMM Run Time (Seconds)				
6	6,583.44	2,801.82	3,007.82	3,015.04	4,165.19
12	6,712.70	3,527.78	4,274.42	4,061.61	4,517.18
24	1,635.46	988.75	740.70	736.69	914.61
	ADD Run Time (Seconds)				
6	192.18	64.23	64.21	73.18	85.86
12	188.81	71.47	55.17	74.51	62.70
24	43.25	35.95	37.23	39.22	44.02
	NORM Run Time (Seconds)				
6	16.54	11.45	10.96	16.16	11.29
12	13.18	8.50	11.05	11.35	12.90
24	13.57	10.97	11.53	8.05	7.44
	MVM Run Time (Seconds)				
6	8.80	3.46	5.06	4.02	7.25
12	10.08	3.13	6.36	7.11	4.66
24	3.56	1.77	2.08	1.77	1.87

Table 14: Effect of "NumMidDimSplits" on MLLib GMM Performance (8 Nodes)

	GMM Run Time (Seconds)				
1	2,466.73	1,440.77	1,391.03	1,477.93	1,361.21
500	2,308.37	926.07	739.91	630.80	761.83
1000	2,224.66	927.77	1,198.63	1,124.36	918.35

Table 15: Comarison of Matrix Addition Methods

Method	Run Time (Seconds)				
Native	722.58	615.02	413.82	593.22	646.10
Custom	159.37	38.30	46.11	47.27	28.24

Table 16: Effect of Number of Segments on MADLib Performance

	GMM Run Time (Seconds)				
6	1,234.10	1,097.47	1,153.95	1,165.01	1,208.63
12	--	--	--	--	--
24	2,321.03	2,182.65	2,129.74	2,175.40	2,095.39
	ADD Run Time (Seconds)				
6	251.41	213.41	143.87	137.61	186.11
12	--	--	--	--	--
24	321.93	263.73	282.65	291.18	251.34
	NORM Run Time (Seconds)				
6	40.08	59.03	47.08	51.94	81.89
12	90.17	106.00	86.98	144.68	98.37
24	95.64	94.69	97.86	105.18	94.55
	MVM Run Time (Seconds)				
6	216.97	203.94	227.01	231.20	209.47
12	265.36	247.24	248.48	244.74	253.37
24	236.66	241.99	247.83	244.80	244.49

Figure 10: Multi-Node Dense Data for MAT with varying D.R.

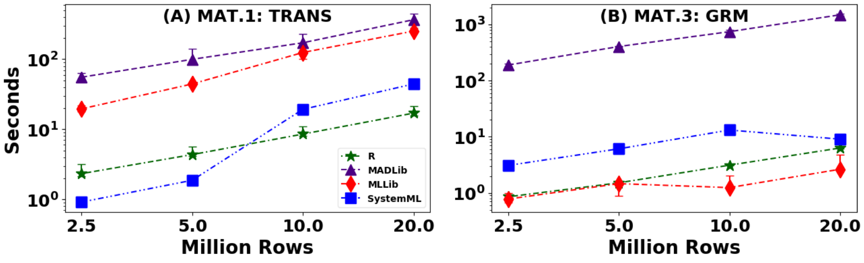


Figure 11: Multi-Node Sparse Data for MAT with varying D.S.

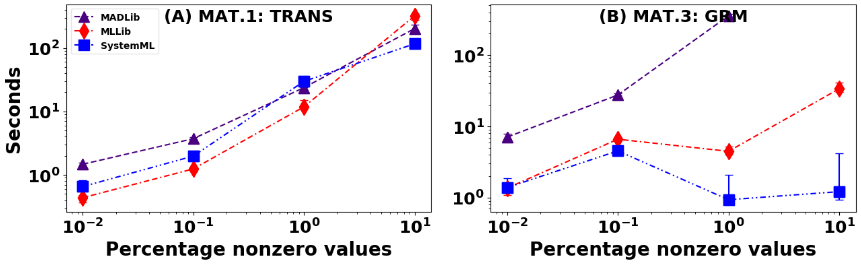


Figure 12: Multi-Node Dense Data for MAT with varying C.N.

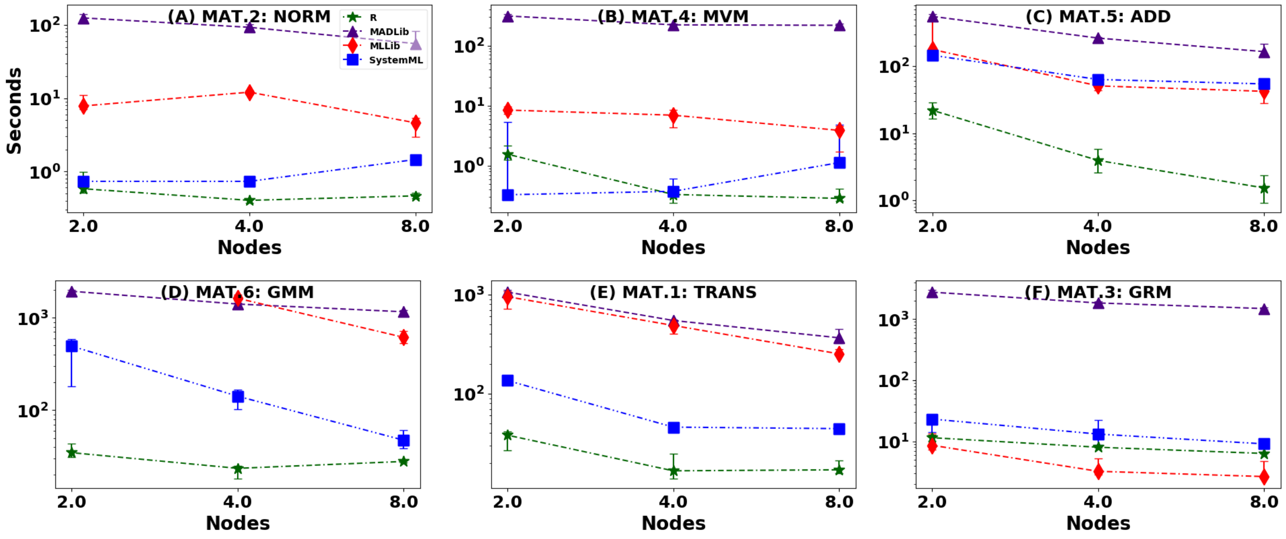


Figure 13: Multi-Node Sparse Data for MAT with varying C.N.

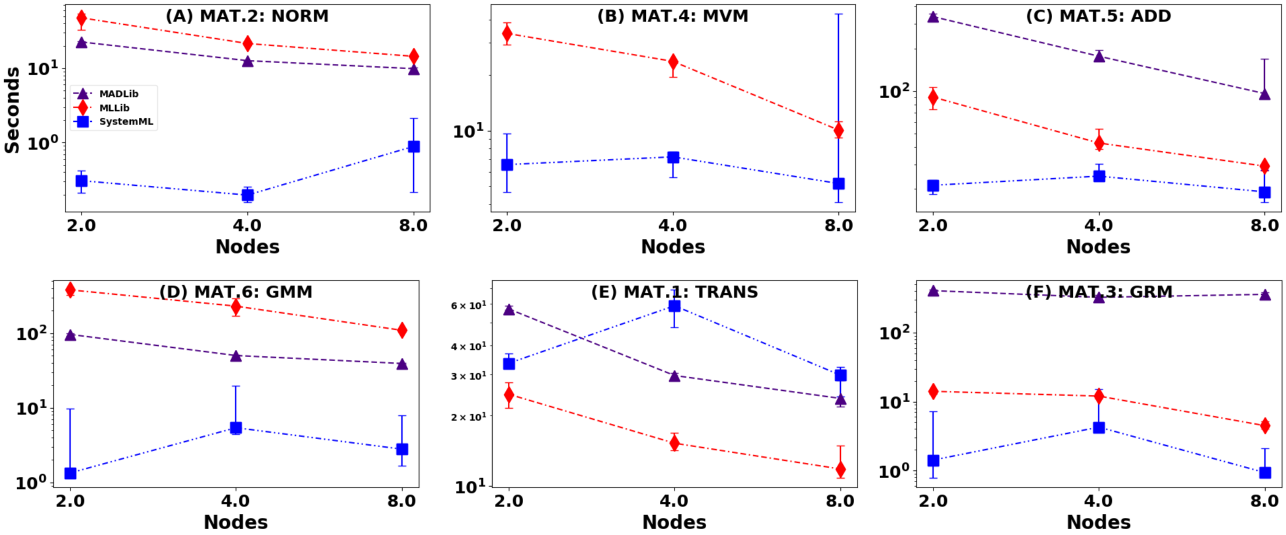


Figure 14: Single-Node Dense Data for MAT with varying D.R.

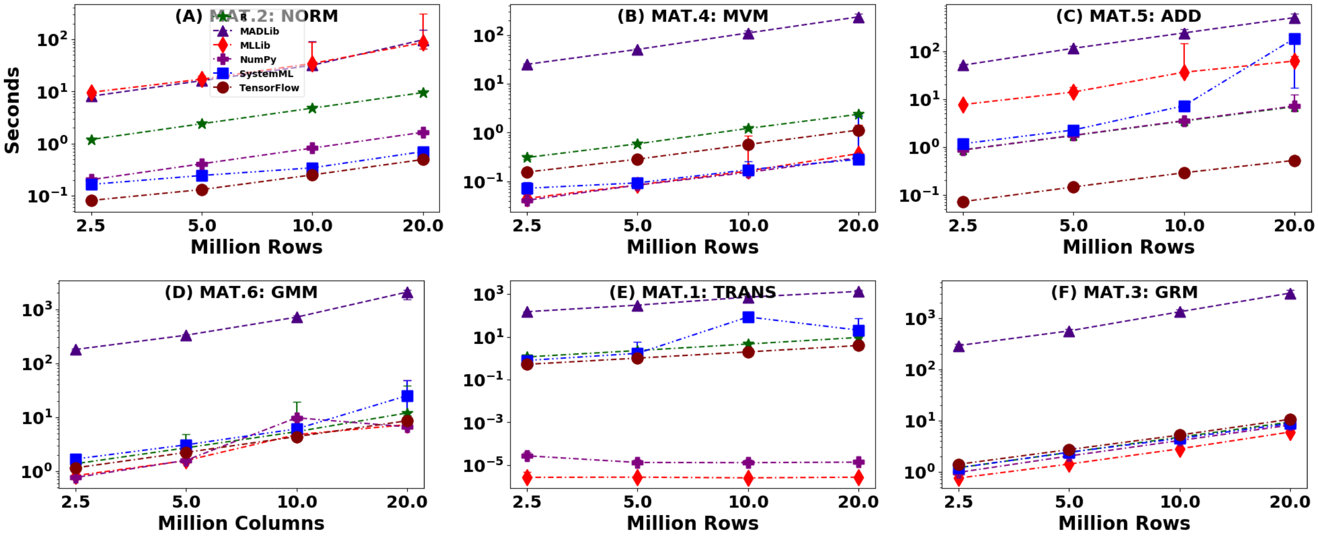


Figure 15: Single-Node Dense Data for ALG with varying C.C.

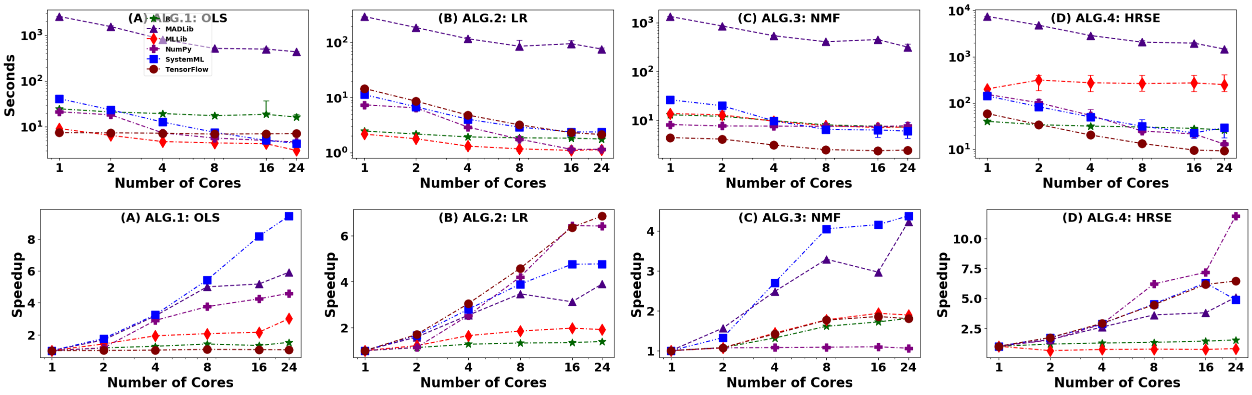


Figure 16: Single-Node Dense Data for MAT with varying C.C.

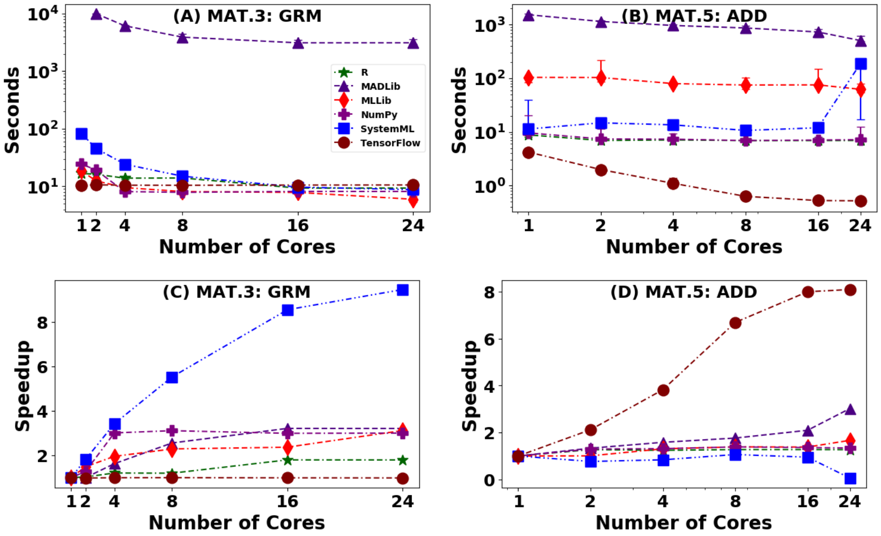


Figure 17: Multi-Node Sparse Data for ALG with varying D.S.

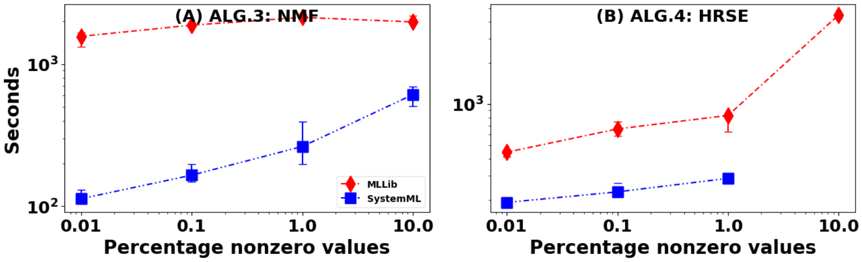
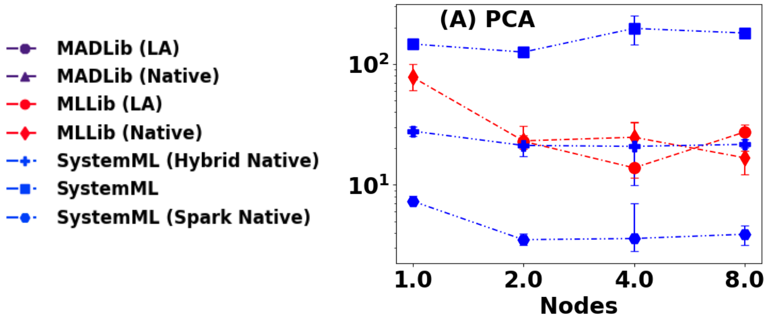


Figure 18: Multi-Node *DenseCriteo* for LA-based and native implementations of ALG.3


MADLib (LA)
MADLib (Native)
MLLib (LA)
MLLib (Native)
SystemML (Hybrid Native)
SystemML
SystemML (Spark Native)
(A) PCA
10^2
10^1
Nodes