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

Anthony Thomas Arun Kumar
University of California, San Diego
{ahthomas, arunkk}@eng.ucsd.edu

ABSTRACT

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

1. INTRODUCTION

Supporting large-scale statistical and machine learning (ML) algorithms over structured data has become a mainstream requirement of modern data systems [6, 22]. Consequently, both the database and cloud service industries are scurrying to make it easier to integrate ML algorithms and frameworks such as R with data platforms [1, 7, 11, 12, 14, 36, 52]. While a roster of canned implementations might suffice for popular existing algorithms, many application users, especially in the enterprise and scientific domains often create custom analysis algorithms for their data. Such users typically write their algorithms using *linear algebra* (LA) notation and are 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, 40]. 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 [84], EMC's MADlib [52], Oracle R Enterprise [14], IBM's SystemML [36], Microsoft's Revolution R [12], Spark MLlib [67] and SparkR [16], Apache Mahout Samsara [73], LA on SimSQL [63], 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. Almost all of them aim to scale LA operations that process the whole dataset *in bulk*, i.e., in one go in a data-parallel manner. A common example of an algorithm that can be expressed in bulk LA is least-squares linear regression (Algorithm 1). This access pattern differs from the access pattern of stochastic gradient descent-based training of neural networks, which is not a bulk LA operation, but rather repeatedly samples data subsets called mini-batches [68]. Thus, except for TensorFlow, the other systems above do not (yet) offer strong support for neural networks. Accordingly, the focus of this paper is on systems for bulk LA workloads.

While the recent activity on scalable LA systems has led to many interesting research papers and open-source systems, a fundamental practical question remains largely unanswered: *From a comparative standpoint, how effective and efficient are such systems for scalable LA and LA-based ML?* By effectiveness, we mean how "easy" they are to use in terms of the amount of code needed to express algorithms using their LA abstractions, as well as their setup logistics. By efficiency, we mean how "fast" they run in different settings, especially the increasingly popular distributed memory setting. Different systems have different assumptions and 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 regimes where they perform well, leaving users in the dark on where they may not perform so well. Additionally, most systems were not compared to strong baselines such as pbdR (powered by ScaLAPACK) [71]. 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.

In this work we take a step towards filling this gap by introducing a suite of performance tests for scalable LA systems and performing a comprehensive empirical comparison of several popular and state-of-the-art LA systems using our tests. Our test suite is inspired by the long line of

work on benchmarks for LA packages in both the single-node in-memory and distributed settings, e.g., benchmarks comparing BLAS, Eigen, Intel MKL, LAPACK, and ScaLAPACK [32,34,43,44,60,69]. 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 *computation environment*. For data scale, we vary both the number of examples and sparsity of the data matrix. For the computation environment, we focus on commodity CPU clusters (as of this writing, almost none of the compared systems offer easy integration with GPUs, FPGAs, or other hardware accelerators in the distributed setting). We vary the number of CPU cores and cluster nodes.

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

We compare the following systems: MADlib, MLlib, SystemML, ScaLAPACK, SciDB, TensorFlow, R, and NumPy. We picked them primarily due to their popularity but also because they are open source and have active user communities online. Our experimental study has revealed numerous interesting behaviors at large data scales, including unexpected tuning headaches, hidden scalability bottlenecks, unusual relative performance trends, and even bugs in some 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. Overall, we find that SystemML offers perhaps the best balance between physical data independence and performance at scale among the compared systems.

Based on our 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 relatively poor; this is a pressing issue due to the ubiquity of large-domain categorical features in ML applications. We also find that the compared systems often 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 combining data and task parallelism could make such systems more useful for crucial meta-level ML tasks such as hyper-parameter tuning [57]. But offering 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 we a suite of tests spanning the axes of task complexity, data scale, and computational environment.
- Using our tests, we perform an extensive empirical study comparing MADlib, MLlib, SystemML, ScaLAPACK, and SciDB in the distributed memory setting, along with TensorFlow, R, and NumPy on a single node. Apart from synthetic data of various scales, we create two benchmark versions of a large real-world dataset from an online advertising firm, Criteo [24].
- We analyze and distill our results into a concrete set of guidelines for practitioners interested in using such systems. We also identify a handful of major open questions for further research on such systems.
- Our findings have already resulted in bug fixes and feature earmarks for SystemML due to our conversations with its developers. We are also speaking with the developers of some of the other systems to help them improve their systems using our results.

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

Outline. Section 2 provides some background on LA systems, 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

A data matrix $\mathbf{X}_{n \times d}$ has n examples and d features. Linear algebra (LA) is elegant and expressive formal language to capture linear transformations of $\mathbf{X}$ (vectors and scalars are special cases). Numerous statistical and ML algorithms can be expressed using LA operations that are "bulk" vectorized computations over $\mathbf{X}$, including such popular algorithms as ordinary least squares (OLS) linear regression, logistic regression solved with many gradient methods [70], non-negative matrix factorization, k-means clustering, and more [40,46,48]. Such algorithms are common in enterprise analytics, as well as social and physical sciences.

Scalable LA systems aim to let users write LA-based algorithms but scale them to larger-than-memory datasets *transparently*, i.e., with some degree of *physical data independence*. Essentially, such systems abstract away lower-level systems issues such as parallel computation, data communication, and fault tolerance (in some cases). In a sense, scalable LA systems aim to achieve for LA what RDBMSs achieve for relational algebra (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, RASDAMAN). We also skip industrial tools such as Oracle R Enterprise and Microsoft Revolution R for legal reasons but hope that our work helps spur more industrial standardization in this arena. We study two popular system environments: single-node in-memory and distributed memory (with disk spills allowed), but focus on the latter. For the former, R and Python's NumPy are 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 [34,71], the array database SciDB [37], the distributed RDBMS-based MADlib [52], and Spark-based MLlib (or SparkML) [67] 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 library treat matrices as first class citizens, provide a rich set of built-in LA operations and algorithms, and are widely used by academics and enterprise data scientists [9]. Most LA operations in R and NumPy are thin wrappers around highly optimized LAPACK and BLAS routines [32,41]. R and Python also provide robust visualization/plotting libraries (e.g., ggplot and matplotlib). Both R and Python are interpreted and dynamically typed, which makes holistic optimization of LA scripts challenging.

pbdR (ScaLAPACK). ScaLAPACK extends LAPACK to distributed memory by re-implementing many LA operations and algorithms using a block partitioning scheme for matrices [34]. It follows the "same program multiple data" paradigm in which one 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.

The "Programming with Big Data in R" (pbdR) library provides higher level R interfaces to ScaLAPACK and OpenMPI [71]. The "distributed matrix" package in pbdR overloads several built-in LA operations in R to enable transparent distributed execution. Conventional R scripts may be run interactively in an interactive command line client called a "Read-Evaluate-Print-Loop" or "REPL" for short. In contrast, pbdR compiles programs into MPI batch jobs that are then submitted for execution on an MPI cluster [47].

MADlib. MADlib is a library that implements both LA primitives and popular ML algorithms over PostgreSQL and the parallel RDBMS Greenplum [52]. Matrices are stored as tables in the DBMS and many matrix operators are implemented directly in SQL, enabling MADlib to exploit the RDBMS for memory management and scalability. Certain matrix operations like eigen or singular value decompositions require the matrix to fit in single-node memory [64]. Related to MADlib (but not compared here) are RIOT-DB [84], which translates LA scripts written in R into SQL, and SimSQL [63] which implements custom datatypes for block partitioned matrices and user-defined functions which implement LA operations.

Spark MLlib/SparkML. MLlib (and the newer SparkML) are libraries that provide LA primitives and popular ML algorithms implemented on Spark RDDs. We focus on MLlib, 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. The user must select a matrix type explicitly and the operators provided by each type are not consistent. For example, the type supporting matrix-matrix multiplication does not support decompositions and vice-versa. All matrix types support either sparse or dense data, but some operators densify sparse matrices.

SystemML on Spark. Introduced for Hadoop and then ported to Spark, SystemML is perhaps the most mature scalable LA system [36]. We focus on the recommended Spark version [3]. SystemML offers a "declarative" language called 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 a block-partitioned format implemented as a Spark RDD. Inspired by RDBMSs, SystemML has an optimizing compiler that converts a DML script to Spark jobs by applying a suite of logical LA-specific rewrite optimizations and physical execution optimizations. The compiler translates a DML script into a physical execution plan which selects operator implementations based on data and system characteristics (e.g. size, sparsity, RAM). SystemML can transparently shift between local and distributed execution.

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 but does not provide scalable LA operators. A TF program is modeled

as a graph in which nodes represent operations over arrays ("tensors") and edges represent dataflow. A TF program is executed in two phases: first the graph is defined, then a node in the graph (not necessarily terminal) is run 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.

SciDB. SciDB is a data management system for multidimensional arrays that was designed primarily for the scientific computing community [37]. SciDB provides a declarative interface similar to SQL in which users compose algebraic operators that can read, write, and update multidimensional arrays. Internally, SciDB shards a single logical array into a set of possibly overlapping chunks that are distributed over one or more instances, possibly residing on different physical nodes. At the time of this writing, SciDB supports a fairly limited set of LA operators, viz., general matrix-matrix multiplication and singular value decomposition. Furthermore, these operators do not support disk spills and require that data fit within distributed memory.

3. DESCRIPTION OF TESTS

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

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

Axis 2: Data Scale Factors. Varying key properties of the data matrix lets us stress test how 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, number of columns and the fraction of non-zero cells, also called *sparsity*. We note that the number of rows and columns governs the shape of the data matrix $\boldsymbol{X}$. We focus on varying the number of rows and data sparsity as these parameters govern most practical workloads.

Axis 3: Computational Scale Factors. This axis captures the amount of computational resources available. Once again, for the sake of simplicity and uniform treatment across all systems, we focus on commodity CPU-based compute clusters in the cloud. Thus, we only vary two settings: number of CPU cores and number of worker nodes. While this is a popular setting for many enterprises and domain scientists, we note that GPUs, FPGAs, and custom ASICs (e.g., TPUs [54]) 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 [83].

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. $\boldsymbol{\epsilon}$ is the residual vector from OLS.

Test Description	Test Semantics
MAT: Matrix Operators ∼ **Vary**: Nodes, Cores, Rows, Sparsity	
Matrix Transpose (TRANS)	$\mathbf{X}^T; \mathbf{X}^T_{i,j} = \mathbf{X}_{j,i}$
Frobenius Norm (NORM)	$\|\mathbf{X}\|_F = \sqrt{\sum_i \sum_j \|\mathbf{X}^2_{i,j}\|}$
Gram Matrix (GRM)	$\mathbf{X}^T\mathbf{X}; (\mathbf{X}^T\mathbf{X})_{i,j} = \sum_k \mathbf{X}_{k,i} \cdot \mathbf{X}_{k,j}$
Matrix-Vector Multiplication (MVM)	$\mathbf{Xw}; (\mathbf{Xw})_i = \sum_k \mathbf{X}_{ik} \cdot \mathbf{w}_k$
Matrix Addition (ADD)	$\mathbf{M} + \mathbf{N}; (\mathbf{M} + \mathbf{N})_{i,j} = \mathbf{M}_{i,j} + \mathbf{N}_{i,j}$
Matrix Multiplication (GMM)	$\mathbf{MN}; (\mathbf{MN})_{i,j} = \sum_k \mathbf{M}_{i,k} \cdot \mathbf{N}_{k,j}$
PIPE: Pipelines and Decompositions ∼ **Vary**: Rows	
Multiplication Chain (MMC)	$\mathbf{X_1X_2X_3}$
Singular Value Decomposition (SVD)	$\mathbf{U\Sigma V}^T \leftarrow \mathbf{X}$
ALG: Bulk LA-based ML Algorithms ∼ **Vary**: Nodes, Cores, Rows, Sparsity	
Ordinary Least Squares Regression (OLS)	$(\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$
Logistic Regression (LR)	See Algorithm 2
Non-negative Matrix Factorization (NMF)	See Algorithm 3
Heteroscedasticity-Robust Std Errors (HRSE)	$(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\text{diag}(\boldsymbol{\epsilon}^2)\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}$

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 metadata to determine the lowest cost plan. Thus, we include this simple pipeline as a test workload. Perhaps surprisingly, most popular scalable LA systems do not optimize this pipeline; the user has to manually fix the multiplication order, which could be tedious in general.

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

3.1.3 ALG: Bulk LA-based ML Algorithms

This is the highest level of task complexity: algorithms expressed as bulk LA scripts. For tractability sake, we include only a handful of popular algorithms that satisfy the following desiderata. First, we 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. First is ordinary least squares (OLS) for linear regression solved using the normal equations [46]. Second is logistic regression (LR) for binary classification solved using batch gradient descent [70]. Third is non-negative matrix factorization (NMF) for feature extraction solved using weighted multiplicative updates [49]. Fourth and final is White's heterscedasticity-robust standard error estimator (HRSE) for OLS, which is common in the domain sciences and enterprises for handling heteroscedasticity [81]. Among these four algorithms, HRSE 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, 46]. Given the data matrix $\mathbf{X}_{n\times d}$ and target vector $\mathbf{y}_{n\times 1}$ with values in $\mathbb{R}$, OLS computes the projection of $\boldsymbol{y}$ onto the space spanned by $\boldsymbol{X}$. OLS is trivial to implement in LA systems, since it has a closed form analytical solution using the pseudo-inverse of $\mathbf{X}$. Algorithm 1 presents the LA script for OLS.

Algorithm 1: Ordinary Least Squares (OLS).

Inputs: $\mathbf{X}$, $\mathbf{y}$

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

Logistic Regression (LR). LR is the most popular algorithm for classification [9] and is typically solved using iterative algorithms expressed in LA. 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 often converge in fewer iterations [70]. 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.

Algorithm 2: Logistic Regression (LR) with BGD.

Inputs: $\mathbf{X}$, $\mathbf{y}$, I: number of iterations, α: step size

for $i = 1$ to I **do**

  $\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \mathbf{X}^T \left(\frac{1}{1+e^{-\mathbf{X}\mathbf{w}}} - \mathbf{y} \right)$

return w

Non-negative Matrix Factorization (NMF). NMF factorizes $\mathbf{X}_{n \times d}$ into two lower rank matrices such that the reconstruction error is minimized. NMF is common for feature extraction, especially on on sparse data (e.g., in text mining). NMF can also use many optimization algorithms; weighted multiplicative updates is popular [33, 61]. Algorithm 3 presents its LA script. This algorithm stress tests matrix-matrix multiplication, including multiplication chains.

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 (SE) of OLS coefficients are used for hypothesis tests about the sign and magnitude of the coefficients in many domain sciences and enterprise settings. This often requires addressing heteroscedasticity–a condition that occurs when the variance of the target is correlated with feature values and may lead to incorrect SEs. To mitigate this issue, the procedure of [81], shown in Algorithm 4, is often used. While the LA notation is succinct, expressing this computation in lower-level code could be tedious and painful. In spite of the popularity of such procedures, among the systems compared, only MADlib offers a robust OLS variance estimator natively. This affirms the need for scalable LA abstractions to enable data scientists to scale such LA-based procedures easily. The HRSE algorithm also tests for several LA optimizations such as avoiding the materialization of the large (but ultra-sparse) matrix $\text{diag}(\boldsymbol{\epsilon}^2)$.

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 [78]. In particular, we focus primarily on the distributed memory setting here, which means most of our discussion centers on pbdR, MADlib, MLlib, and SystemML. In general, the data for our experiments will fit fully in distributed memory and disk spills should not be necessary. For a handful of tests, intermediate results *may* exceed distributed memory depending on the execution plan selected by the system. All the code 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`.

Table 2: Software Packages and Versions

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

Experimental Environment. All experiments were run on the CloudLab "Clemson" site using the `c6320` instance type [72]. Each physical node has the following hardware specifications: two Intel E5-2683 v3 14-core CPUs (2.00 GHz), 256GB ECC memory, two 1 TB 7.2K RPM 3G SATA HDDs, and a dual-port Intel 10Gbe NIC (X520) network adapter. On top of these nodes, we use OpenStack to create and manage a virtual cluster. Scripts to create and manage clusters are available on the project GitHub page. Each OpenStack node runs Ubuntu 16.04 and has 200 GB of RAM, 24 CPUs, and 700 GB of disk. Table 8 describes the versions of software packages used.

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

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

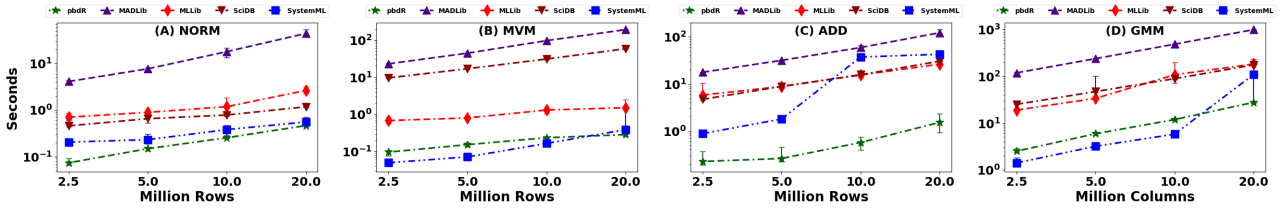


Figure 1: Multi-Node Dense Data for MAT Varying Rows

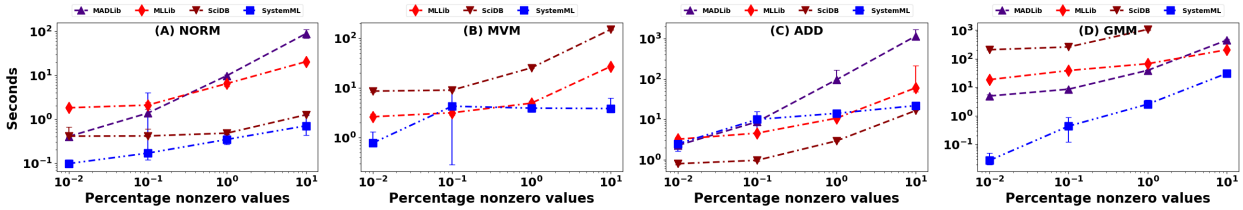


Figure 2: Multi-Node Sparse Data for MAT with Varying Sparsity

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 [78]. We believe our system tuning efforts are reasonable from the standpoint of *typical data scientist users*. While more extensive tuning of a given system might improve its runtimes slightly, that is orthogonal to our larger goal of a unified comparative understanding. We hope our work spurs more interest among the developers and users of such systems to apply our tests for further empirical analysis.

4.1 Results for MAT

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

All systems except MADlib finish in under 5s on norm and matrix-vector multiplication. MADlib's LA operations are slower than the others systems', primarily because it has higher per-tuple RDBMS processing overheads and the I/O time of materializing output tables. SystemML is almost always faster than MLlib, MADlib and SciDB, since SystemML mostly pulls computation into its driver and executes operators in local mode. SystemML's runtimes see a marked rise on matrix-addition at 10 million rows and GMM at 20 million rows; based on the execution logs, we verified that this rise happened because computation got moved to Spark at this data scale. In Spark mode, SystemML exhibits comparable performance to MLlib, which always runs in distributed mode. Thus, SystemML's ability to transparently switch between local and distributed execution based on its data size estimates is effective. Finally, pbdR exhibits strong performance on all four operators in spite of always executing in distributed mode. This underscores both the overhead imposed by the other scalable LA systems and the importance of including a strong baseline system when evaluating performance.

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

Single-Node Dense Data Vary Cores. We fix n at 20 million and d at 100, which is about 16GB. We vary the number of cores on a single node to study multicore speedup behaviors. Figure 3 presents the results for two key LA operations. We see that the benefit of additional cores typically plateaus at between 4 and 16 cores depending on the system.

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

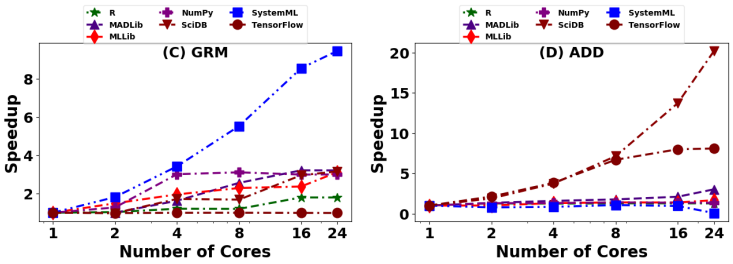


Figure 3: Single-Node Dense Data for MAT with Varying Cores

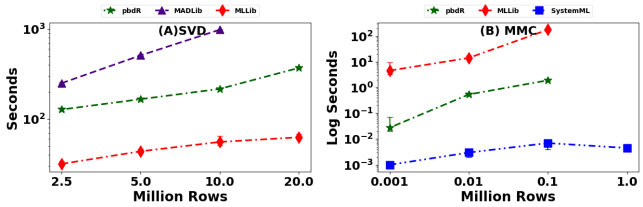


Figure 4: Multi-Node Dense Data for PIPE with Varying Rows

Speedup behavior varies considerably by both operator and by system. For SciDB, the strong speedup in the ADD operation can be explained by increased parallelism from adding more SciDB instances. For the in-memory systems, multicore speedups are largely dictated by the underlying LA implementation. For example, R's implementation of ADD never uses multiple threads while TensorFlow's does. Such implementation decisions are likely reflective of the different workloads targeted by these systems. R was designed with small to medium size data in mind. In such a context, overhead from setting up threads and copying data may exceed the benefits of parallelism for very simple binary operators like addition. Finally, we note that while TensorFlow's runtimes are generally fastest, it performs all computations using single precision floating point numbers (`float`) by default, while the other systems all use double precision numbers (`double`).

4.2 Results for PIPE

Back to the 8-node setting, we now run matrix multiplication chain (MMC) and SVD on dense data. Panel (A) presents results for SVD. For SVD we fix $d = 100$ and vary n. We compute only the singular vectors corresponding to the ten largest singular values. SystemML is not shown because it does not provide a distributed SVD. SciDB is not shown because its SVD implementation must compute all three factor matrices independently and so a fair comparison is not possible. MLlib's strong performance for this operator is due to a novel algorithm (described in [82]), which is optimized for map-reduce.

Panel (B) presents results for MMC. We omit MADlib and SciDB as the user must specify the matrix-multiplication order explicitly. As the plot shows, only SystemML is able to determine 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 modestly large n. We note, however, that neither pbdR not MLlib claim to be able to detect this optimization; thus, this represents an avenue for improving these systems.

4.3 Results for ALG

Multi-Node Dense Data Vary Rows. For the same 8-node setup and data scales as Figure 1, we run the **ALG** tests (LA-based ML algorithms) for dense data. For LR and NMF, reported runtimes are the average over three iterations. Figure 5 presents the results. Trends for this plot (and explanations thereof) largely follow those of Figure 1 with the relative gap in performance between SystemML and MLlib larger than for the individual matrix operators. As was the case in Figure 1, SystemML performs most computations in the driver program, which explains its high performance, often surpassing pbdR. The jump in runtime for SystemML on HRSE at 10 million rows is again caused by switching to Spark mode. For pbdR, it was necessary to hand-optimize the sparse diagonal matrix multiplication in HRSE to avoid out-of-memory errors. MADlib does not automate composition of LA operators, which forces all intermediate results to be written to disk, and in turn, leads to high I/O costs and runtimes for iterative algorithms. SciDB does not provide an operator for matrix inversion or solving a linear system. To implement OLS and HRSE on SciDB, we were forced to compute $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ using SVD. Furthermore, to compute the inverse, it is necessary to fully compute $SVD(\boldsymbol{X}^T\boldsymbol{X})$ three times as the SVD operator provided is strangely only able to return a single matrix in the decomposition per call.

Single-Node Dense Data Vary Cores. For the same single-node setup and data scales as Figure 3, we run the **ALG** tests for dense data to study the multicore speedup behaviors of all systems. To obtain tractable runtimes for MADlib, we reduce matrix size to 2.5×10^6. Figure 14 presents the results. Speedup behavior varies substantially by system and algorithm. R is faster than NumPy for LR and HRSE, but not on the others, likely due to the overheads of intermediate data copies for matrix-matrix multiplication [76]. Surprisingly, MLlib is comparable in performance to TensorFlow on OLS and LR. But it is slower on NMF and much slower on HRSE due inefficiency in the method used to compute the dense-diagonal product in HRSE, which we were forced to code by hand. We linked MLlib with a native BLAS library because it yielded substantial performance improvements over the default JVM implementation. For TensorFlow, we used a special "loop" operator provided by the TF API, since it does not require us to copy data between NumPy and TF at each iteration.

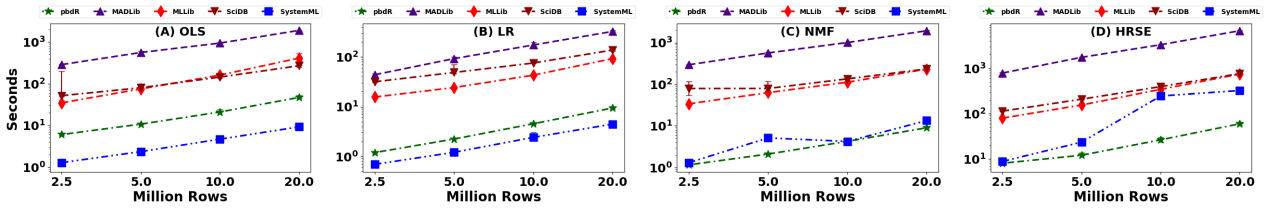


Figure 5: Multi-Node Dense Data for ALG with varying Rows

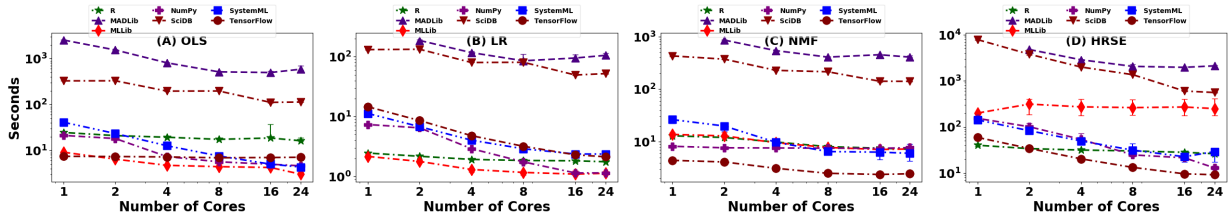


Figure 6: Single-Node Dense Data for ALG with varying Cores

4.4 Scalability on Criteo Datasets

While our above results with synthetic data already give interesting insights about the compared systems, we also want to know how these systems perform on a large real-world dataset for ML analytics. In particular, we are also interested in comparing the native, non-LA based implementations offered by some systems against their own LA-based versions. We use a large public dataset from Criteo, whose full size is 1 TB [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 16(A) presents the results. We observe three major trends overall. First, across the board, the native implementations of LR and OLS have much lower per-iteration runtimes than the LA-based implementations (except for SystemML on LR). Thus, the scalable LA abstractions, especially on MADlib and MLlib, come at a high performance penalty, which users should be aware of. Second, at the largest setting of 8 nodes, all three systems have comparable performance with their native implementations, while SystemML is the fastest for LA-based implementations. Both of these trends suggest that users might be better off using the native implementations of well-known algorithms such as LR and OLS and resort to the scalable LA abstractions only if 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.

Multi-Node *SparseCriteo* Vary C.N. Figure 16(B) presents the results. This plot has much fewer lines than Figure 16 (A) 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 Gram-matrix 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

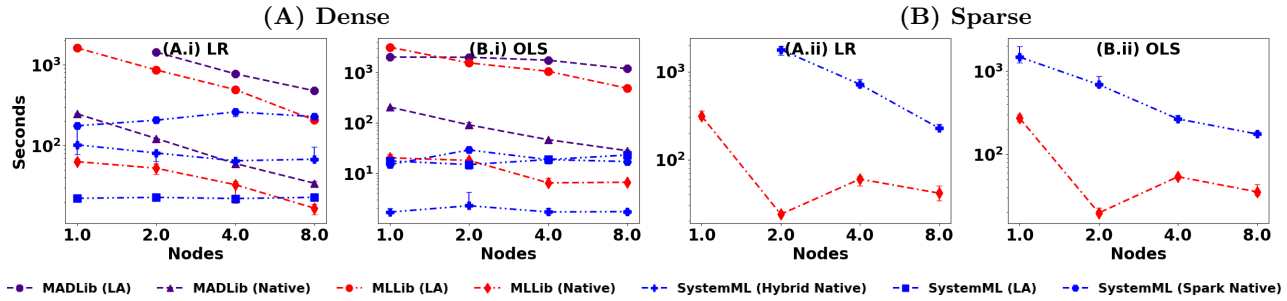


Figure 7: Multi-Node *Criteo* for LA-based and native implementations of LR and OLS

Table 3: Tokens for Implementation of ALG Tests.

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

but its runtimes were much higher than MLlib's and SystemML's respective native implementations, which obfuscated the trends. The large drop in Spark runtime between 1 and 2 nodes is because Spark must use disk cache in the single node configuration while the multi-node configurations can use memory only.

5. ANALYSIS AND DISCUSSION

We discuss key lessons learned from our comparative evaluation. We begin by discussing our experience implementing the ML algorithm tests in each system. We then provide concrete guidance to practitioners about the strengths and weaknesses of each LA system, followed by a summary of the key takeaways from our empirical analysis. Finally, we discuss several key open research questions related to the design and implementation of scalable LA systems.

5.1 Algorithm Implementation Effort

To evaluate the implementation effort required for writing the **ALG**, one should ideally conduct a user study with real-world data scientists to obtain a thorough picture of system usability. In lieu of such an extensive study, we provide a rough quantification by tabulating the number of "tokens" necessary to implement an algorithm in each system. We do not include "non-essential" tokens such as calls used to induce computation by Spark.

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

5.2 Guidelines for Practitioners

We here summarize each scalable LA system's strengths and weaknesses. We organize our discussion around the practical difficulties we faced when implementing the ML algorithms in 3.1.3 and specifically address the following issues. How difficult is 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? We elide discussion of the single-node tools (R, NumPy, and TensorFlow) as these tools are generally well known to practitioners.

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

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

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

MADlib. We found tuning MADlib on Greenplum to be relatively simple but laborious. There is little concrete advice on tuning the number of Greenplum segments per node [8]. Determining the optimal number of segments required manually re-initializing and loading the database which was labor-intensive. MADlib has generally good support for sparse data and was competitive on the **MAT** tests with sparse data. However, it suffers from a serious scalability bottleneck due to a hard limit of 1 GB imposed on the maximum size of an array by PostgreSQL. MADlib's use of an RDBMS gives it a reasonable level of physical data independence, since data partitioning and caching are abstracted away. However, interleaving table creation in SQL and LA computations might be highly unintuitive for many ML-oriented data scientists. The RDBMS offers excellent support for data pre-processing using SQL, but flexibility for writing feature engineering pipelines is poorer than Spark. Greenplum supports fault tolerance and automates disk spills (except for the array type). MADlib's native ML implementations were typically significantly faster than LA-based implementations and were competitive with the other systems compared. Overall, MADlib is well suited to users with large RDBMS-resident datasets and who are happy with canned ML algorithm implementations.

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

5.3 Key Takeaways

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

(1) Transparently switching between execution in local and distributed mode improves performance and reduces user effort. Recent surveys of data scientists show that a majority work with datasets that could fit on a beefy commodity node's memory [10, 35]. By providing APIs that work mostly seamlessly in both local and distributed mode, SystemML attained good performance on smaller datasets without requiring us to re-implement code for scaling it to large datasets.

(2) Automatically detecting LA optimizations such as dense-diagonal matrix multiplication (as in the HRSE test) and reordering matrix multiplication chains yields performance benefits and reduces user effort. Once again, only SystemML detected such LA-specific optimizations automatically. All the other systems forced us to hand optimize multiplication orders, which often required trudging through their documentation.

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

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

5.4 Open Research Questions

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

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

Auto-tuning Data Layout and System Parameters. The promise of physical data independence means users should not have to expend much effort for tuning data layout and system parameters. Yet, most systems required us to tweak one or more of such parameters; some just crashed

without such tuning. For instance, MLlib required manually tuning partitioning and caching commands, while MADlib required several data layout decisions for interleaving LA scripts with table creation in SQL. Such low-level decisions are likely to be unintuitive for statistical or ML-oriented users. MLlib and TensorFlow also require lower-level programming skills in Python (or Scala for MLlib) to resolve such issues. SystemML offers perhaps the most automation of such decisions, but its dependence on Spark necessitates some memory-related tuning. Overall, more work is still needed to achieve true physical data independence for scalable LA. Extending the lessons of auto-tuning from the RDBMS and MapReduce worlds (e.g., [53, 79]) to scalable LA is another avenue for new research.

Multi-node "COST" and Parallelism Models. A surprising takeaway relates to the bedrock of database systems: multi-node data parallelism. Conventional wisdom has that using many cheap nodes is likely better than fewer expensive beefy nodes, even for dozens of GBs. But as our Criteo results show, the speedup curves flatten quickly, at least for the compared systems. This issue is the multi-node version of the "COST" factor in [65]. It is not clear if this issue will be mitigated at larger scales (TBs or PBs), since that would require even more nodes to get more total memory (at least for SystemML and MLlib). In turn, that could raise communication costs for distributed LA operations, as well associated metadata and query processing overheads. Furthermore, recent surveys of data science practitioners show that about 70% analyze datasets under only 100 GB [10, 35]. Thus, while faster multi-node data-parallel LA implementations are useful, we think a more pressing research challenge is to support transparent scalability for *task parallelism* across nodes combined with data parallelism within a node. Such support is useful for crucial meta-level ML model selection tasks such as hyper-parameter tuning [35, 55, 57, 75]. A recent Spark-TensorFlow integration does support task-parallel hyperparameter-tuning for neural networks but only for small datasets that can be broadcasted [4]. A related research question is determining the optimal cluster size for a given dataset and LA script, similar to how [75] sizes clusters for a few specific ML algorithms.

Including ML Accuracy for Evaluation. The faster performance of MLlib's and MADlib's native ML implementations compared to their LA-based versions suggests that including ML accuracy as a criterion could substantially alter the relative performance landscape. But it is non-trivial to standardize comparisons of both runtimes and ML accuracy *simultaneously* because the latter is inextricably tied to hyper-parameter tuning, which is usually a non-smooth and non-convex meta-level optimization problem [75]. A given system will likely fall on multiple points on the accuracy-runtime *Pareto frontier* for different datasets, ML algorithms, and hyperparameter-tuning choices, which makes apples-to-apples comparisons between systems hard. Moreover, different implementations of the same ML task could have differing hyper-parameters. For instance, how does one standardize how hyperparameter-tuning should be done when comparing, say, MADlib's SGD-based LR with SystemML's native LR? Should sub-sampling be allowed to enable more extensive hyper-parameter tuning? We leave such questions to future work but note that there is growing interest in Pareto frontier-based benchmarks, at least for deep learning tasks, although they do not yet account for the costs of hyper-parameter tuning [17].

6. OTHER RELATED WORK

Benchmarks of LA Packages. LA packages such as BLAS, LAPACK, ScaLAPACK, and Eigen have been extensively benchmarked [32, 34, 43, 44, 60, 69]. For instance, the LINPACK benchmark focuses on the efficiency (measured in MFLOPs per second) of solving a system of linear equations [42, 59]. 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 [59] 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 [45, 58]. 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 [76].

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, 52, 67, 77], 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 [73]. [66] compared SciDB [37], Myria [80], Spark, and TensorFlow for a specific scientific image processing task but not for general LA or ML workloads. [39] compare Spark, GraphLab [62], SimSQL [38], 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] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

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

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

[45] 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.

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

[47] 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.

[48] 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.

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

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

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] 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.

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

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

[62] 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.

[63] 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.

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

[65] 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.

[66] 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.

[67] 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.

[68] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

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

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

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

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

[73] 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.

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

[75] 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.

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

[77] R. Taft, M. Vartak, N. R. Satish, N. Sundaram, S. Madden, and M. Stonebraker. Genbase: A complex analytics genomics benchmark. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 177–188. ACM, 2014.

[78] 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`.

[79] 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`.

[80] 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.

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

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

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

[84] 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

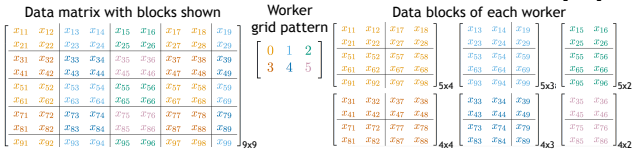
Table 4 presents a summary of salient characteristics of each system in the slate we compare. We additionally provide example implementations of OLS in each of these systems. Note that LOC counts may not agree exactly with the numbers reported in table 3 in the body. The example programs here have been 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. 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, 41]. 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.

Program 2 shows the implementation OLS 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.

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 8 gives an example of the block-cycling partitioning scheme from [74] for a 9×9 matrix with a grid of 6 workers and blocks of size 2×2.

Figure 8: Block-cyclic matrix partitioning [74].



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 [71]. 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 [47].

SystemML 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.

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. 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].

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. MADLib is a library that implements both LA primitives and popular ML algorithms over PostgreSQL and the parallel RDBMS Greenplum [52]. 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 [50]. 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 [64]. Related to MADlib is RIOT-DB [84], which avoids SQL as a front-end and opts for the so-called "query generation" approach [56]. 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 [84]. Also related is the recent SimSQL [63], which relies on custom user-defined datatypes for block-partitioned matrices and custom user-defined functions that implement LA operations.

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. 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 [51]. 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.

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.

SciDB. Program 7 presents an implementation of OLS in SciDB's "array query language." There are several noteworthy points about this program. First, it is much more verbose than the other programs. This is because SciDB does not provide an operator for solving a linear system or inverting a matrix directly. This necessitates the use of the singular value decomposition to compute the inverse. Furthermore, because SciDB operators must return only a single table, the decomposition must be performed three times to compute each constituent matrix! We observe as well that the `gemm` operator for matrix-multiplication computes

$MN + Q$. The final matrix Q is not optional, and the user must materialize a table of zeros to compute the standard matrix product.

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 tuned these parameters by starting with a single large executor which was given all memory and CPU available and then dividing the memory and CPU allocated to each executor by two until performance stopped improving. We found that using 3 cores and roughly $20GB$ of RAM per executor led to the best performance. With these settings, Spark allocated eight executors per node. Results from this mini-benchmark are presented in **??**.

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.

SciDB. The key tuning considerations for SciDB are the number of instances used per node and the size of the sub-blocks into which arrays are partitioned. Based on our results in figure 3 we found that 24 instances (1 instance per core) was generally optimal in the single node setting. In the eight node cluster, we found that using such a large number of instances led to substantial overheads. The SciDB manager instances must broadcast queries to each worker instance. This can lead to significant overheads on large clusters. We found that 8 instances per cluster was optimal, and replicated a benchmark available on the SciDB website to validate our tuning settings. SciDB requires that matrix blocks be square for GMM and so we compared block sizes of 100, 500 and 1000 and found 1000 to be generally optimal.

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))
```

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).

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 [72] "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

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)

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');
```

Program 7: OLS in SciDB

```
store(gemm(X, X, Z, transa:true), XTX);

store(gemm(project(apply(cross_join(
  transpose(gesvd(XTX, 'VT')) as V,
          project(apply(
                  gesvd(XTX, 'S'), sigma_inv,
                  POW(sigma,-1)), sigma_inv)
              AS SINV, V.i, SINV.i),
              vsinv, v*sigma_inv), vsinv),
          transpose(
      gesvd(XTX, 'U')), Z), XTX_INV);

gemm(XTX_INV, gemm(X, y, Z, transa:true), Z)
```

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 cluster 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" [47] 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 9 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 10 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

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

computation in the driver.

Multi-Node Dense Data Vary C.N. Figure 11 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 12 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 13 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 [76] 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 15 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

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

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 14 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 **??** 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 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 16 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 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 Settings on MLLib Performance

Memory (GB)	Cores	**GMM** Run Time (Seconds)				
200	24	5,746.80	2,982.14	2,900.22	624.84	243.63
100	12	836.57	290.92	276.03	269.36	245.12
50	6	826.93	342.80	243.16	235.86	255.95
25	3	764.11	230.12	257.69	269.57	257.46

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

Figure 9: Multi-Node Dense Data for MAT with varying D.R.

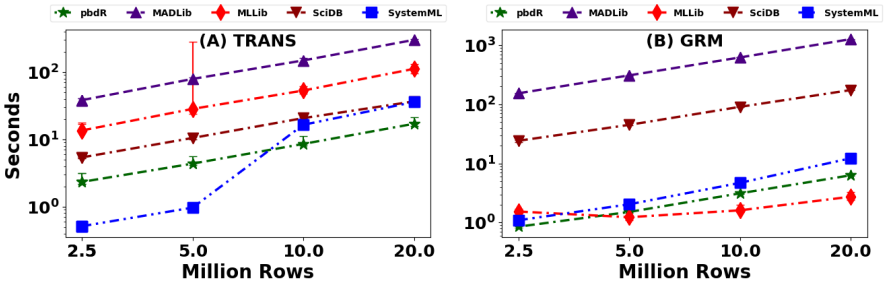


Figure 10: Multi-Node Sparse Data for MAT with varying D.S.

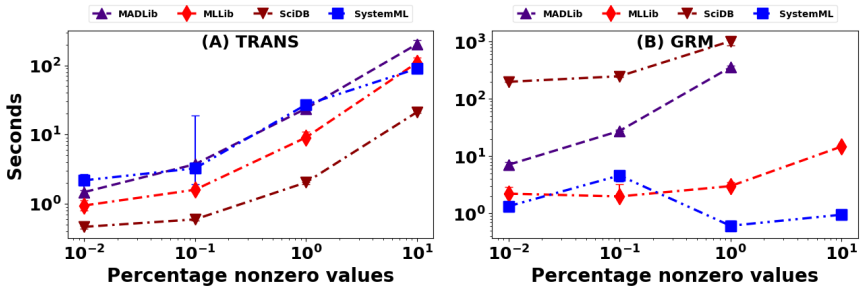


Figure 11: Multi-Node Dense Data for MAT with varying C.N.

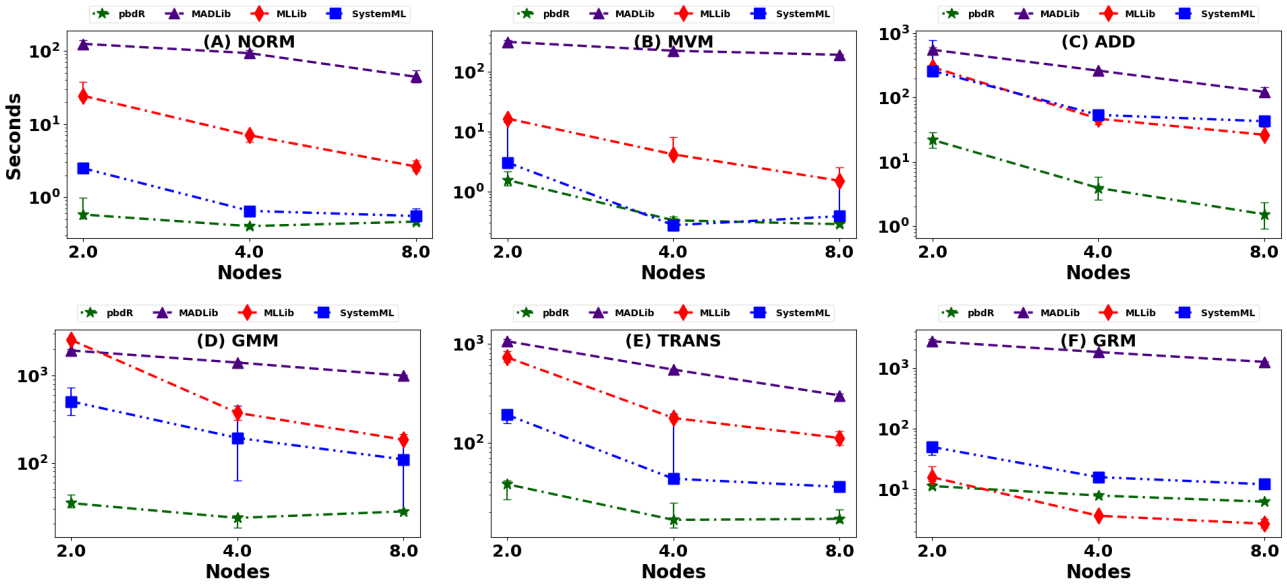


Table 15: Comarison of Matrix Addition Methods

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

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

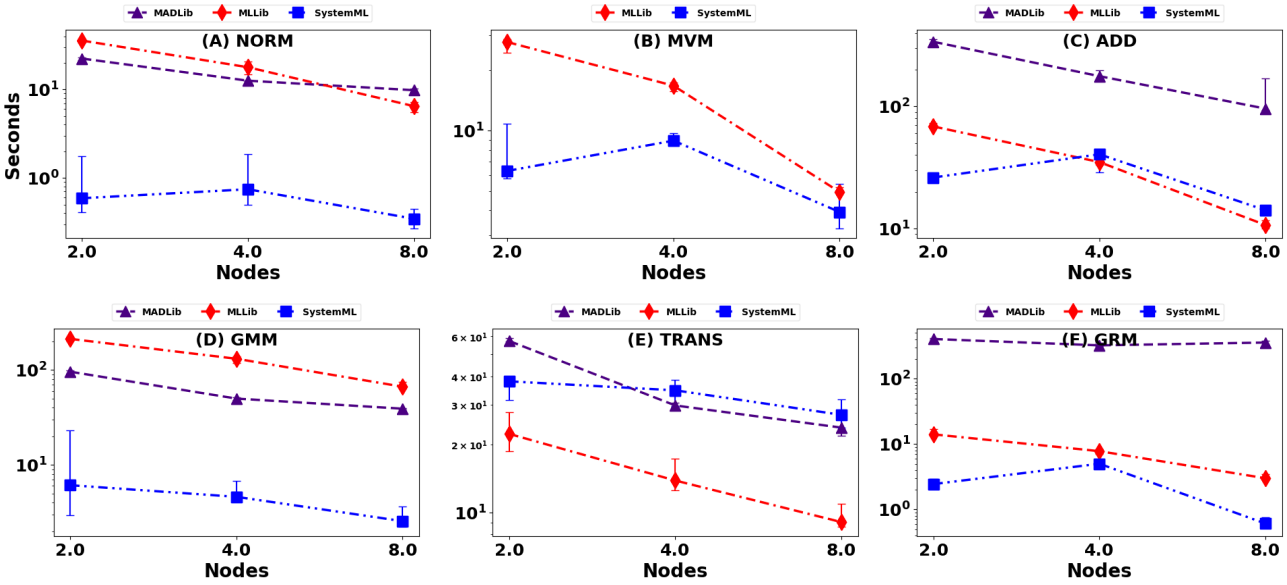


Figure 13: Single-Node Dense Data for MAT with varying D.R.

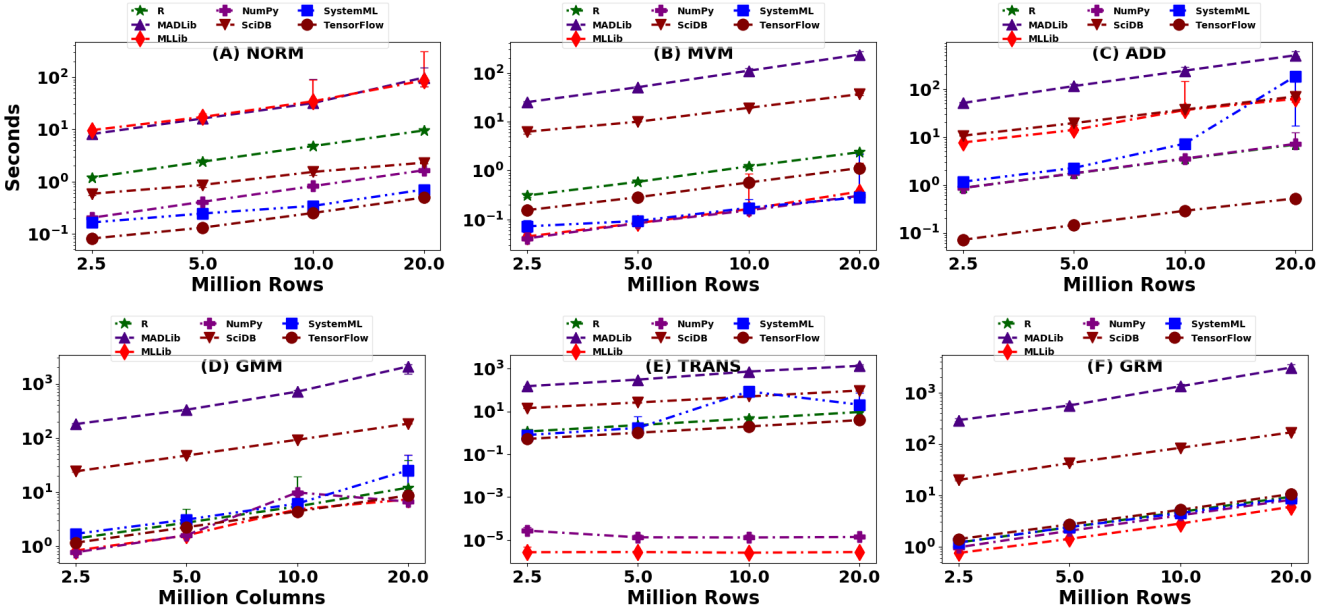


Figure 14: Single-Node Dense Data for ALG with varying C.C.

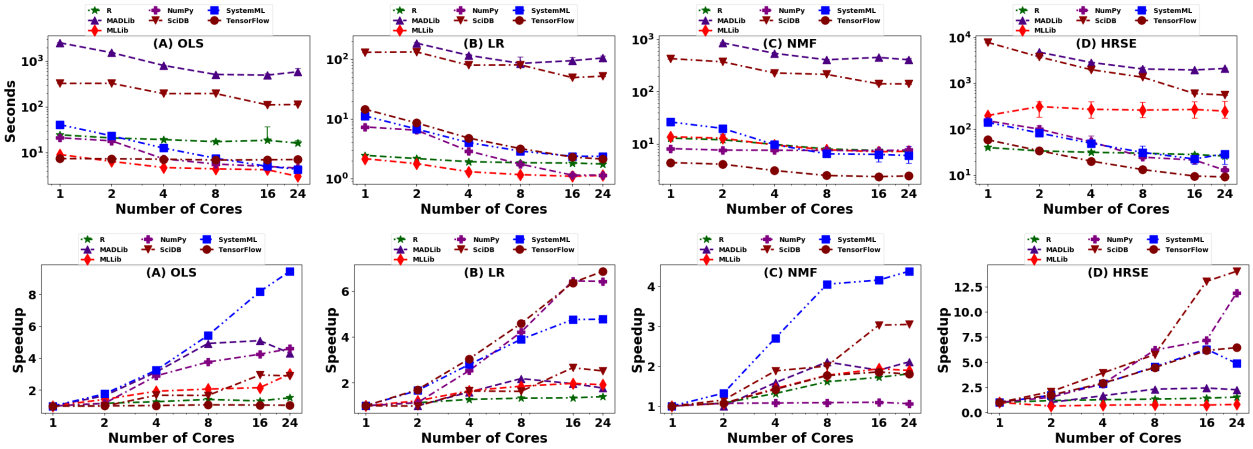


Figure 15: Single-Node Dense Data for MAT with varying C.C.

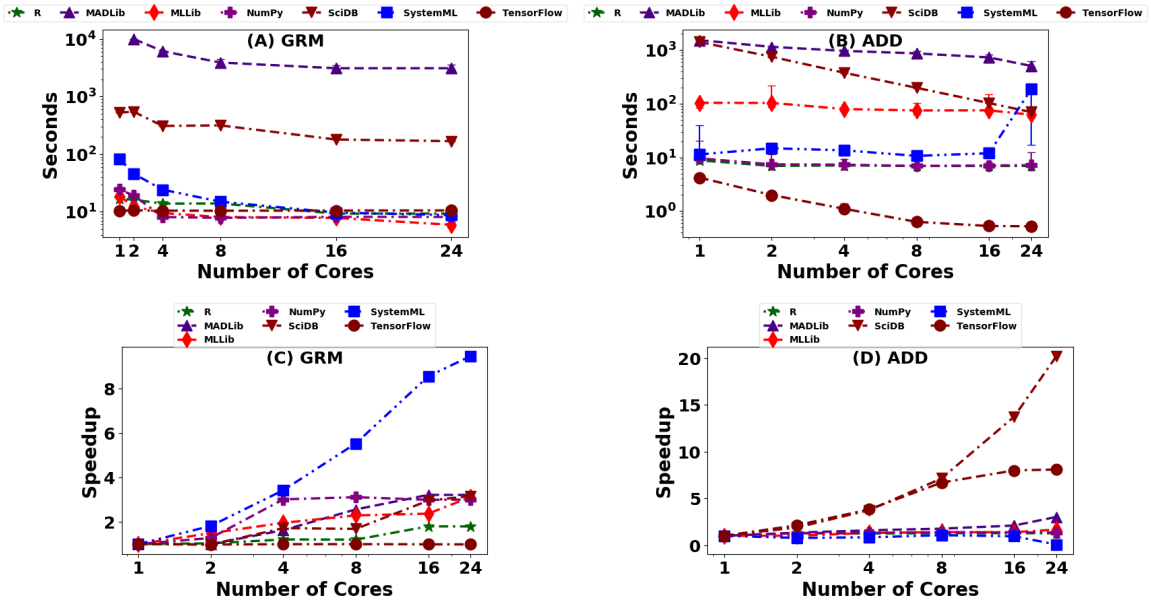


Figure 16: Multi-Node *DenseCriteo* for LA-based and native implementations of ALG.3

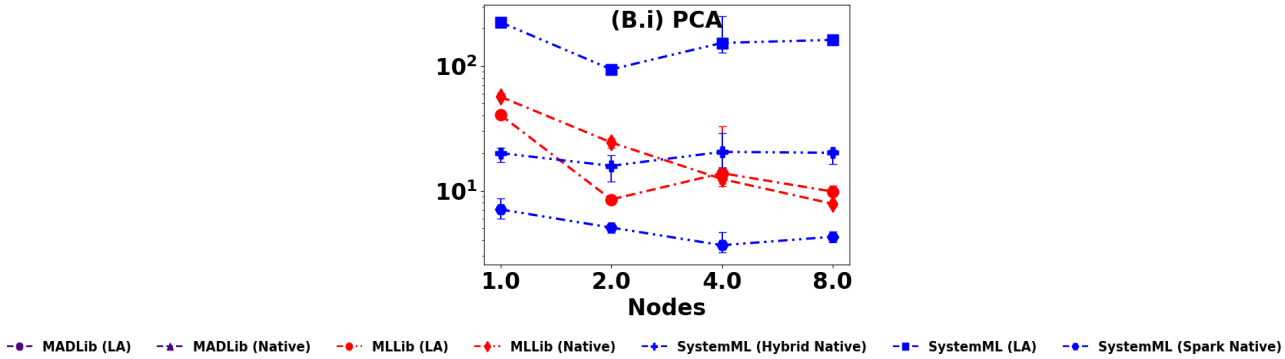


Table 16: Effect of Number of Segments on MADLib Performance

	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