# Samsara: Declarative Machine Learning on Distributed Dataflow Systems

**Sebastian Schelter**
Technische Universität Berlin
`sebastian.schelter@tu-berlin.de`

**Andrew Palumbo**
`apalumbo@apache.org`

**Shannon Quinn**
University of Georgia
`spq@uga.edu`

**Suneel Marthi**
`smarthi@apache.org`

**Andrew Musselman**
`akm@apache.org`

## Abstract

We present *Samsara*, a domain-specific language for declarative machine learning in cluster environments. Samsara allows its users to specify programs using a set of common matrix abstractions and linear algebraic operations, similar to R or MATLAB. Samsara then compiles, optimizes and executes these programs on distributed dataflow systems. The aim of Samsara is to allow mathematicians and data scientists to leverage the scalability of distributed dataflow systems via common declarative abstractions, while drastically reducing the need for detailed knowledge of the programming model and execution scheme of the underlying systems. Samsara is part of the Apache Mahout library and supports backends like Apache Spark and Apache Flink. In this paper, we introduce the concepts of Samsara, showcase its compilation and optimization techniques using a simple example, and experimentally evaluate some of the benefits of these optimizations.

## 1 Introduction

A critical component of modern large-scale machine learning (ML) is operating on, accessing, and analyzing datasets stored in distributed filesystems running on a cluster of machines. In such an environment, data analysis is often conducted using distributed dataflow engines, that allow for scalable, data-parallel execution of programs. Examples of such systems include Apache Spark [24], Apache Flink [1] and H2o [12]. Unfortunately, these systems are difficult to program, as their programming model is heavily influenced by the underlying data-parallel execution scheme. Usually, programs consist of a sequence of parallelizable second-order functions (such as `map`, `reduce` or `groupBy`) that dictate how the system should execute user-defined first-order functions on partitioned data [3, 24]. Such programming models are non-intuitive for users without a background in distributed systems, and are in general hard to program without a detailed understanding of the underlying execution model. Furthermore, the available programming abstractions typically rely on partitioned, unordered sets; this is a mismatch for ML applications that mostly operate on linear algebra constructs (e.g., vectors and matrices). Therefore, implementing distributed linear algebra operations on dataflow systems is a tedious and difficult task.

This problem is partially resolved by the emergence of easy-to-use ML libraries such as *MLlib* for Apache Spark [17]. However, a library is not sufficient for all kinds of ML users, as many advanced users need to create new algorithms and adapt existing ones, especially while prototyping and conducting exploratory data analysis. Samsara targets exactly these use cases by providing a lightweight domain-specific language (DSL) in Scala that offers different matrix primitives and supports a wide set of linear algebraic operations. Users decide on the physical placement of matrices (e.g., in the memory of the driver machine or distributed in the aggregate main memory of the cluster), and write programs by chaining transformations on these matrices (Section 2). The resulting programming model is declarative: users specify what operations to execute on these matrices, but leave the decisioning on how to physically execute them to the system. Therefore, the resulting programs are short and concise, and resemble programs written in R or MATLAB. Samsara compiles user programs to the underlying distributed dataflow systems, conducts logical optimizations, and chooses the physical execution strategy at runtime (Section 3).

The contributions of this paper are the following:

- We present a lightweight declarative domain-specific language to define ML programs which consist of matrix transformations with linear algebraic operations. This DSL allows users to retain control over the physical placement of data and operations, and thereby enables hybrid programs where parts execute on distributed dataflow systems (Section 2).
- We showcase how to logically optimize the resulting programs, and how to choose physical execution strategies for the resulting operator graphs (Section 3).
- We experimentally evaluate the benefits of the proposed optimizations (Section 5).

## 2 Overview

**Domain-Specific Language**. We give a brief overview of the language constructs in Samsara which are required for the subsequent example. We refer the interested reader to [15] and [16] for a complete reference. Samsara is implemented in Scala and distinguishes between in-memory and distributed data structures. Samsara offers different tensor types for in-memory data: dense vectors, two implementations of sparse vectors (optimized for random access as well as sequential access), dense matrices, several flavors of sparse matrices, and implementations of matrices with special characteristics (e.g., diagonal matrices). These in-memory data structures support common tensor operations and allow random reads and writes to the underlying data. For distributed data, Samsara offers a special kind of matrix, the so-called *Distributed Row Matrix (DRM)*. A DRM is a row-wise partitioned matrix, where the partitions are stored in the main memory of the cluster machines. For performance reasons, all rows of a partition are usually merged into a matrix block. DRMs offer a reduced set of operations (e.g., they do not allow random writes), which will be executed in parallel on the cluster. DRMs seamlessly work together with in-memory matrices and vectors. Furthermore, DRMs allow mapreduce-style programming, e.g., modifying matrix blocks through custom user-defined code with a so-called `mapBlock` operation and aggregating blocks with an `allReduceBlock` operation. The Mahout library contains a number of distributed algorithm implementations built on the Samsara DSL, e.g., naive bayes text classification [18], cooccurrence analysis for recommendations [19, 8], matrix factorization with Alternating Least Squares [25], QR factorization, stochastic singular value decomposition [13], and stochastic principal components analysis.

**Architecture and Execution**. Figure 1 illustrates the architecture of Samsara. Applications are written using the Scala DSL. The in-memory operations are immediately executed, while operations on DRMs are deferred. The system records the actions to perform on these distributed matrices, and internally builds a directed acyclic graph (DAG) of logical operations from them, where vertices refer to matrices and edges correspond to transformations between them. Materialization barriers (e.g., persisting a result or collecting a matrix into local memory) implicitly trigger execution. Upon execution, the DAG of logical operators is optimized and transformed into a DAG of physical operators to execute. These physical operators are specific to one of the backends that Samsara supports (currently Apache Spark and Apache Flink), and run the distributed parts of the program using the respective backend.

**Example: Distributed Ridge Regression**. Our running example throughout this paper will be the common task of predicting a continuous target variable $\mathbf{Y}$ from a feature matrix $\mathbf{X}$ by ridge regression. For our example, we assume that $\mathbf{X}$ is a tall and skinny matrix. We aim to compute the parameters $\boldsymbol{\beta}$ of the regression by solving the normal equation: $\beta = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{Y}$. In our implementation, we exploit the fact that $\mathbf{X}^\top\mathbf{X}$ as well as $\mathbf{X}^\top\mathbf{Y}$ live in the column space of $\mathbf{X}$ with small dimensionality (as $\mathbf{X}$ is tall and skinny). This allows us to fit $\mathbf{X}^\top\mathbf{X}$ into the memory of the driver for a moderate number of columns, since this will only require memory quadratic in the number of columns of $\mathbf{X}$. Analogously we can also easily fit $\mathbf{X}^\top\mathbf{Y}$ into the driver memory.

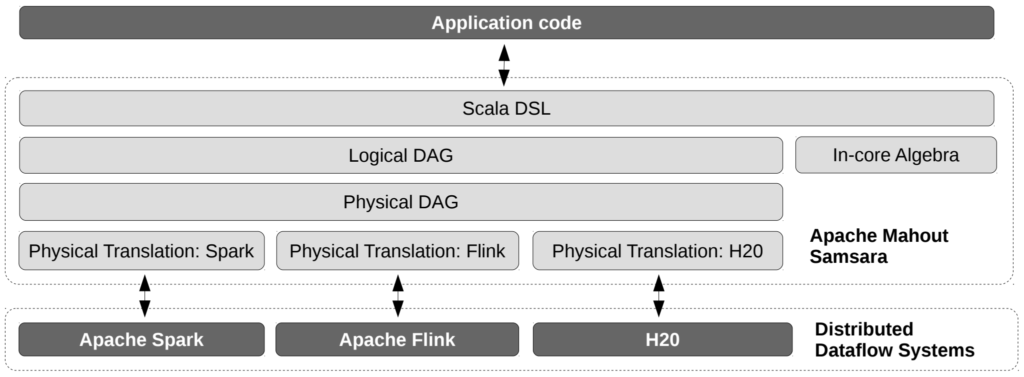


Figure 1: Architecture.

The resulting Samsara program is shown in Listing 1. The code defines a function `dridge` which takes two arguments: a distributed data matrix `data` containing the features and the target variable (in the right-most column), as well as the regularization parameter $\lambda$ (line 1). We form the feature matrix $\mathbf{X}$ by slicing out all columns except for the right-most one from the input data matrix, and append a column of ones for the bias term (line 3). We then slice out the right-most column of the input data matrix to retrieve the target $\mathbf{Y}$ (line 4), and compute $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{Y}$ via distributed matrix multiplications (lines 6). Finally, we materialize $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{Y}$ on the driver (via the `collect` calls in lines 9 and 10), add the regularization term to the diagonal of $\mathbf{X}^\top\mathbf{X}$ (line 12), and use an in-core solver to compute the parameter estimate $\boldsymbol{\beta}$ (line 14).

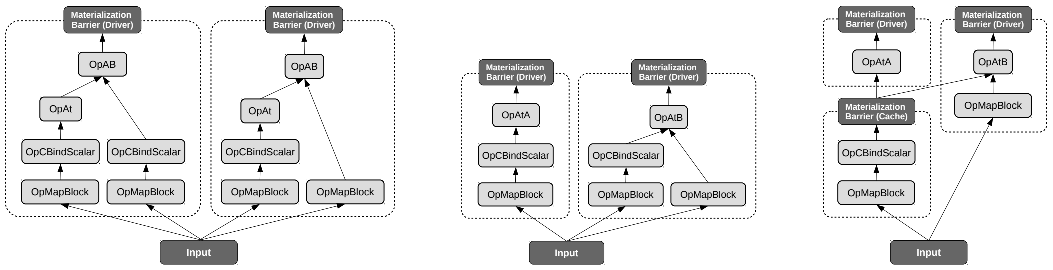
```
1  def dridge(data: DrmLike[Int], lambda: Double): Matrix {
2    // slice out features, add column for bias term
3    val drmX = data(::, 0 until data.ncol) cbind 1
4    val drmY = data(::, data.ncol) // slice out target

6    val drmXtX = drmX.t %*% drmX // distributed matrix
7    val drmXtY = drmX.t %*% drmY // multiplications

9    val XtX = drmXtX.collect // fetch results
10   val XtY = drmXty.collect // into driver memory

12   XtX.diagv += lambda // add regularization

14   solve(XtX, XtY) // compute parameters in-core on driver
15 }
```

Listing 1: Distributed Ridge Regression for tall & skinny matrices using Samsara.

## 3 Runtime & Optimization

Next, we discuss how the program from resulting our example will be optimized and executed. Samsara's optimizer is rule-based with reliance on lazily deduced cost heuristics for the choice of physical operators. The optimization occurs for every operator DAG associated with a pipeline (all the operators between two materialization barriers).

**Logical Optimization**. In the logical optimization phase, Samsara uses pattern matching to derive equivalent plans that produce identical outputs, but potentially result in a shorter runtime. These rewrites aim to reduce the number of passes over the input data, and typically compact the plan by merging operators. Figure 2b shows the final plan to execute produced by optimizing the original plan (shown in Figure 2a) for our example code from Listing 1. The expression `OpAt` $\rightarrow$ `OpAB` which represents the computation of $\mathbf{X}^\top\mathbf{Y}$ from the middle pipeline in Figure 2a is merged into the operator `OpAtB` in the optimized plan illustrated in Figure 2b. The latter operator is a specialized implementation to compute a matrix-transpose-times-matrix multiplication. The next expression that gets optimized


Materialization Barrier (Driver)
OpAB
OpAt
OpCBindScalar
OpMapBlock
Input
OpAtA
OpAtB
Materialization Barrier (Cache)

(a) Logical plan before optimization.

(b) Rewritten logical plan.

(c) Rewritten logical plan with manual caching.

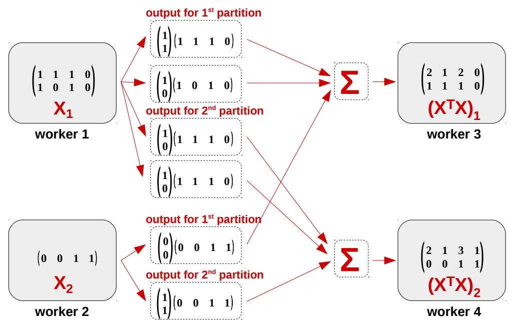
Figure 2: Logical execution plan resulting from the distributed ridge regression code in Listing 1.

is `OpMapBlock` $\rightarrow$ `OpCbindScalar` $\rightarrow$ `OpAt` $\rightarrow$ `OpAB` $\leftarrow$ `OpCbindScalar` $\leftarrow$ `OpMapBlock`. This expression represents the part of the code that slices out $\mathbf{X}$ from the input matrix and subsequently computes $\mathbf{X}^\top\mathbf{X}$. This expression is rewritten in two steps: first, the partial expression `OpAt` $\rightarrow$ `OpAB` is merged to `OpAtB`, so that the full expression becomes `OpMapBlock` $\rightarrow$ `OpCbindScalar` $\rightarrow$ `OpAtB` $\leftarrow$ `OpCbindScalar` $\leftarrow$ `OpMapBlock`. Second, we recognize that both of the inputs to `OpAtB` refer to same matrix, which allows us to replace `OpAtB` with a special operator `OpAtA` for transpose-times-self matrix multiplications. This saves having to compute the result of `OpMapBlock` $\rightarrow$ `OpCbindScalar` $\rightarrow$ `OpAt`, and shrinks the resulting DAG down to `OpMapBlock` $\rightarrow$ `OpCbindScalar` $\rightarrow$ `OpAtA`.

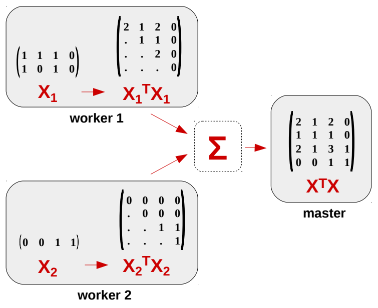
There are situations when multiple materialization barriers are created by the user code. Such evaluations can benefit from using common computational paths. In our example, we compute two expressions, $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{Y}$, and the formation of $\mathbf{X}$ involves reading the input, which incurs some overhead, as we need to access a distributed filesystem each time if the input matrix is not cached. Here, the overall program may benefit from forming $\mathbf{X}$ first and then substitute the result into the $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{Y}$ computations (which is not necessarily true in all cases). However, the optimization triggers for expressions such as $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{Y}$ are placed at different points in a program. At the time when the first expression optimization is triggered, the optimizer is unable to reason about common computational path benefits w.r.t. expressions it has not yet encountered. Therefore our program may benefit from explicitly placing *materialization barriers* in the middle of logical expressions. Not only does such a barrier enforce the materialization of an intermediate result in its entirety, but it also provides an option to place the result into a distributed cache with a desired retention policy (e.g., the Spark cache). In order to achieve that in our example, we would add the statement `drmX.checkpoint(CacheHint.MEMORY_ONLY)` after line 3. The resulting logical plan is illustrated in Figure 2c.

**Choice of physical operators**. After the logical plan has been fixed, Samsara chooses the actual physical operators for the operations to perform. This choice is based on three characteristics of the operands: (1) whether they possess a special structure (e.g., diagonal matrices), (2) their dimensions (e.g., to use special implementations for tall and skinny matrices), (3) their partitioning (e.g., to use local instead of distributed joins for matrices with co-located blocks).

We first discuss the choice of the physical operator for the logical operator `OpAtA` which represents the computation of $\mathbf{X}^\top\mathbf{X}$ in our regression code. Samsara offers two different ways to physically execute this matrix multiplication, both of which are variants of the row-outer-product formulation of matrix multiplication that only requires a single pass over the input matrix. Samsara features a physical operator for general transpose-times-self matrix multiplication (Figure 3a), and another variant that is optimized for a tall and skinny input matrix, illustrated in Figure 3b. In the following, let $(\mathbf{X}_1, \ldots, \mathbf{X}_p)$ denote the $p$ partitions (blocks of rows) of $\mathbf{X}$, such that horizontally stacking these partitions restores the original matrix $\mathbf{X}$; furthermore let $m$ and $n$ denote the number of rows and columns of $\mathbf{X}$, respectively. The first physical operator computes the outer product of every row of $\mathbf{X}$ with itself (using a *map* operation) and sums up the resulting rank-1 matrices with a *reduce* operation, thereby computing $\mathbf{X}^\top\mathbf{X} = \sum_{i=1}^{m} x_{i\bullet}^\top x_{i\bullet}$. In order to control the memory consumed for intermediate results, the computation of the outer product matrices can also be conducted in multiple steps. Figure 3a illustrates this for a step number of two: at first the outer product of the first half of



(a) Distributed computation of $\mathbf{X}^\top\mathbf{X}$ via summation of partial outer products of rows.



(b) Distributed computation of $\mathbf{X}^\top\mathbf{X}$ via summation of local gram matrices.

Figure 3: Physical execution strategies for the distributed computation of a gram matrix.

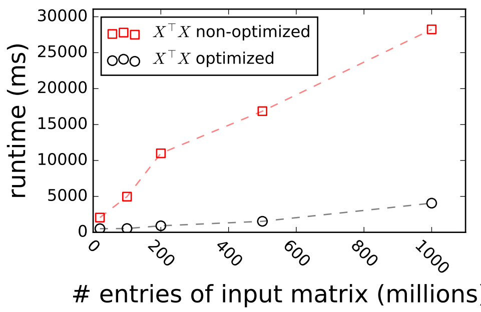
each row vector with itself is computed, subsequently the computation of the outer product of the second half of each row vector with itself is conducted. The second physical operator assumes that we can fit as many matrices of size $(n(n+1))/2$ (the upper triangular part of an $n$ by $n$ matrix) into the memory of a worker machines as we have tasks assigned to it. The actual execution of the operator is illustrated in Figure 3a: Every worker $i$ computes the upper triangular half of $\mathbf{X}_i^\top\mathbf{X}_i$ with regard to its assigned matrix partition $\mathbf{X}_i$. The driver machine collects the results of these computations from the workers and sums them up to produce the final result: $\mathbf{X}^\top\mathbf{X} = \sum_{i=1}^{p}\mathbf{X}_i^\top\mathbf{X}_i$, which it redistributes in the cluster.

Next, we discuss the choice of the physical operator for executing the logical operation `OpAtB` in our example, which corresponds to the computation of $\mathbf{X}^\top\mathbf{Y}$. The physical execution of `OpAtB` avoids the transposition of the left operand. Instead it joins the matrix blocks of both operands on their row keys, multiplies column ranges of the left matrix with the corresponding row blocks of the right matrix and sums up the results. In the choice of the execution strategy for this operator, Samsara's optimizer makes use of its capability to track identical matrix partitioning. This tracking is implemented by assigning a unique identifier to a DRM upon its creation and propagating this identifier through the operators applied to the DRM, given that they preserve the partitioning. In the case of Listing 1, both $\mathbf{X}$ and $\mathbf{Y}$ are derived from the same input matrix and only have operations applied (such as `OpMapBlock`) that preserve the partitioning. So, the optimizer provides the information to `OpAtB` that the operands are identically partitioned. During execution, this allows Samsara to exploit the co-partitioning of the inputs and internally replace the required distributed join with a local join on co-located matrix blocks that does not require to re-partition the operand matrices, which saves a round of network communication.

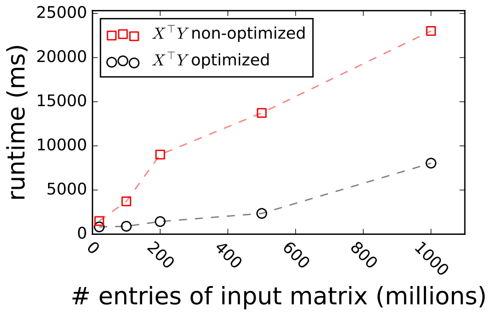
## 4 Related Work

Most similar to our work is the *SystemML* platform [10, 5, 6, 20, 9], which allows it users to write programs in an R-like language, compiles and optimizes these programs, and executes them on distributed dataflow systems. SystemML employs a more holistic optimization than Samsara, as it accesses the abstract syntrax tree of the whole program (including control flow), and takes the decision of where to execute computations (e.g., in-memory or distributed) entirely out of the hands of the user. In comparison, Samsara represents a lightweight approach with a stronger focus on flexibility: it supports non-int keyed matrices, mapreduce-style programming with matrices and makes it easy to integrate any JVM-based library and to extend the system.

*MLlib* [21, 17] is a popular ML library on top of Apache Spark, which also offers several distributed matrix abstractions [7]. However, this work is orthogonal to ours, as the aim of MLlib is to provide highly usable out-of-the box functionality, rather than a declarative environment for custom algorithm implementations. The same is true for *SparkR* [23], which aims to integrate Spark with the R language, but does not provide declarative functionality for distributed linear algebra. A related direction is the deep embedding of the APIs of dataflow systems in their host language [2], where the potential to extend this embedding and the resulting optimizations to linear algebraic operations is currently explored [14].



(a) Effect of the optimizations on the runtime of the distributed computation of $\mathbf{X}^\top\mathbf{X}$.



(b) Effect of the optimizations on the runtime of the distributed computation of $\mathbf{X}^\top\mathbf{Y}$.

Figure 4: Experimental evaluation of the benefits of various optimizations for the regression example.

## 5 Experimental Evaluation

In order to evaluate the benefit of the optimizations discussed in Section 3, we compare the standard execution mode of Samsara for the ridge regression program in Listing 1 with a variant where we manually disable these optimizations[1]. We leverage a cluster running HDFS 2.7.1 and Apache Spark 1.6.2, comprised of 24 machines with 8 cores and 32 GB of main memory each. We synthetically generate tall and skinny input matrices with 20 columns and an increasing number of rows (up to a billion entries), and store them in the distributed filesystem. Next, we measure the runtime of the pipelines executed in our code. We repeat every experiment ten times and report the median runtime. Figure 4a shows the results for the pipeline computing $\mathbf{X}^\top\mathbf{X}$ (the left pipeline in the plan from Figure 2b). We see that the optimized version executes in fewer than four seconds even for an input matrix with a billion entries. The overhead of the non-optimized version is huge and its runtime increases much steeper. This performance difference is expected as the non-optimized variant uses a costly general matrix multiplication operator instead of the specialized transpose-times-self matrix-multiplication operator. Analogously, Figure 4b shows the results for the pipeline computing $\mathbf{X}^\top\mathbf{Y}$ (the right pipeline in the plan from Figure 2b). The optimized variant executes in less than a third of the runtime of the non-optimized variant. The performance difference arises from the fact that the non-optimized variant needs to perform an extra re-partitioning step for the distributed join involved in the computation. We conclude that the results clearly demonstrate the benefits of the proposed optimizer. We also run the experiments with the manual caching optimization described in Section 3, and find this not to decrease the runtime in our setup, as it is cheap to re-compute $\mathbf{X}$ via pipelineable operations.

## 6 Conclusion and Future Work

We presented Samsara, a DSL to define declarative ML programs consisting of matrix transformations, which are optimized and executed on distributed dataflow systems. We discussed some exemplary logical optimizations, as well as the choice of the corresponding physical operators, and experimentally showed their benefits. A current major limitation of Samsara is its lack of native speed for in-core matrix operations, as these are executed in the JVM using code based on the Colt library [4]. To overcome this drawback and provide a bridge to native performance on modern hardware, we are in the process of integrating ViennaCL [22] for selected operators. Another issue are performance problems caused by certain backend specifics, e.g., due to the lack of efficient intermediate result caching in Apache Flink or due to inefficient memory usage in Apache Spark. Here, it would be interesting to explore MPI-based backends, which have been shown to beat JVM-based systems by several orders of magnitude for certain distributed matrix computations [11]. Finally, we acknowledge that Samsara is the result of a community effort from the Apache Mahout project. Among this community, we would especially like to thank Dmitriy Lyubimov for his fundamental contributions to the Samsara design and codebase.

[1]code available at `https://github.com/sscdotopen/mahout-mlsystems/tree/ssc-mlsystems`

## References

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[2] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61. ACM, 2015.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[4] P. Binko, D. Merlino, W. Hoschek, T. Johnson, and A. Pfeiffer. The cern colt library. *URL http://acs. lbl. gov/software/colt/, online*, 2004.

[5] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. Systemml's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

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

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

[8] T. Dunning and E. Friedman. *Practical Machine Learning: Innovations in Recommendation*. O'Reilly Media, 2014.

[9] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment*, 9(12), 2016.

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

[11] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottaalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, et al. Matrix factorization at scale: a comparison of scientific data analytics in spark and c+ mpi using three case studies. *arXiv preprint arXiv:1607.01335*, 2016.

[12] H2o. H2o prediction engine, http://www.h2o.ai/.

[13] N. P. Halko. *Randomized methods for computing low-rank approximations of matrices*. PhD thesis, University of Colorado, 2012.

[14] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: Towards optimization across linear and relational algebra. In *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR '16, 2016.

[15] D. Lyubimov. Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines. `http://mahout.apache.org/users/sparkbindings/ScalaSparkBindings.pdf`.

[16] D. Lyubimov and A. Palumbo. *Apache Mahout: Beyond MapReduce*. 2016.

[17] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[18] J. D. Rennie, L. Shih, J. Teevan, D. R. Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *ICML*, volume 3, pages 616–623. Washington DC), 2003.

[19] S. Schelter, C. Boden, and V. Markl. Scalable similarity-based neighborhood methods with mapreduce. In *Proceedings of the sixth ACM conference on Recommender systems*, pages 163–170. ACM, 2012.

[20] S. Schelter, J. Soto, V. Markl, D. Burdick, B. Reinwald, and A. Evfimievski. Efficient sample generation for scalable meta learning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1191–1202. IEEE, 2015.

[21] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. In *2013 IEEE 13th International Conference on Data Mining*, pages 1187–1192. IEEE, 2013.

[22] P. Tillet, K. Rupp, S. Selberherr, and C.-T. Lin. Towards performance-portable, scalable, and convenient linear algebra. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[23] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. Franklin, I. Stoica, and M. Zaharia. SparkR: Scaling R Programs with Spark. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, 2016.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[25] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *International Conference on Algorithmic Applications in Management*, pages 337–348. Springer, 2008.