In-memory Distributed Matrix Computation Processing and Optimization

Yongyang Yu[*], Mingjie Tang[*], Walid G. Aref[*],
Qutaibah M. Malluhi[†], Mostafa M. Abbas[‡], Mourad Ouzzani[‡]
[*]*Purdue University* [†]*Qatar University* [‡]*Qatar Computing Research Institute, HBKU*
{yu163, tang49, aref}@cs.purdue.edu, qmalluhi@qu.edu.qa, {mohamza, mouzzani}@hbku.edu.qa

***Abstract*—The use of large-scale machine learning and data mining methods is becoming ubiquitous in many application domains ranging from business intelligence and bioinformatics to self-driving cars. These methods heavily rely on matrix computations, and it is hence critical to make these computations scalable and efficient. These matrix computations are often complex and involve multiple steps that need to be optimized and sequenced properly for efficient execution. This paper presents new efficient and scalable matrix processing and optimization techniques for in-memory distributed clusters. The proposed techniques estimate the sparsity of intermediate matrix-computation results and optimize communication costs. An evaluation plan generator for complex matrix computations is introduced as well as a distributed plan optimizer that exploits dynamic cost-based analysis and rule-based heuristics to optimize the cost of matrix computations in an in-memory distributed environment. The result of a matrix operation will often serve as an input to another matrix operation, thus defining the matrix data dependencies within a matrix program. The matrix query plan generator produces query execution plans that minimize memory usage and communication overhead by partitioning the matrix based on the data dependencies in the execution plan. We implemented the proposed matrix processing and optimization techniques in Spark, a distributed in-memory computing platform. Experiments on both real and synthetic data demonstrate that our proposed techniques achieve up to an order-of-magnitude performance improvement over state-of-the-art distributed matrix computation systems on a wide range of applications.**

***Index Terms*—Matrix computation; query optimization; distributed computing**

I. Introduction

In the era of big data, data scientists and analysts often need to analyze large volumes of data in a diverse array of application such as business intelligence applications, self-driving cars, social network analysis, web-search, online advertisement bidding, and recommender systems. Most of the algorithms in these applications are expressed using machine learning (ML) models, e.g., principle component analysis (PCA), collaborative filtering (CF) and linear regression (LR), that involve linear algebra operations and heavy matrix computations as building blocks. Furthermore, many network analysis algorithms are expressed using matrix operations, e.g., PageRank, betweenness centrality, and spectral clustering [21]. Recently, tensor factorization [35] has become a popular model to capture relationships among multiple entities, which also extensively relies on matrix computations. Thus, it is important for these models to have access to an efficient and scalable execution engine for matrix computations. The advent of MapReduce [16] has spurred numerous distributed matrix computation systems, e.g., HAMA [29], Mahout [4], and SystemML [18]. These systems not only provide comparable compute efficiency to widely used scientific platforms [18], e.g., R [6], but also offer better scalability and fault-tolerance. However, these systems suffer from two main shortcomings. First, they are unable to reuse intermediate data [33]. The inability to efficiently leverage intermediate data greatly impedes the performance of further data analysis with matrix computations. In addition, these systems do not leverage the power of distributed memory offered by modern hardware.

One promising way to address the above challenges is to develop an efficient execution engine for large-scale matrix computations based on an in-memory distributed cluster of computers. Spark [33] is a computation framework that allows users to work on distributed in-memory data without worrying about the data distribution or fault-tolerance. Recently, a variety of Spark-based systems for matrix computations have been proposed, e.g., MLI [30], MLlib [5], and DMac [32]. Although addressing several challenges in distributed matrix computation processing, none of the existing systems leverage some of the special features of matrix programs to generate efficient partitioning schemes for matrix data at both the input and intermediate stages. The special features we are referring to, and which are prevalent in ML algorithms, include sparse matrix chain multiplications, low-rank matrix updates, and invariant expressions in loop structures. Since matrix computations are inherently memory intensive, an execution engine that cannot leverage these special features will overwhelm the hardware capacity.

For illustration, consider Gaussian Non-negative Matrix Factorization (GNMF) [24], a widely used ML model for clustering documents and modeling topics of massive text data. The input to GNMF is a $d \times w$ document-term Matrix $\mathbf{V}$, where d corresponds to the number of documents, and w corresponds to the number of terms. Each cell V_{ij} records the frequency of term j in document i. GNMF assumes that Matrix $\mathbf{V}$ can be characterized by p hidden topics such that $\mathbf{V}$ can be factorized into the multiplication of two hidden factor Matrices $\mathbf{W}_{d\times p}$ and $\mathbf{H}_{p\times w}$, i.e., $\mathbf{V} \approx \mathbf{W} \times \mathbf{H}$. In real-world applications, the number of topics p is chosen between 50 and 200. Typically, d and w are much larger than p. For example, in the Netflix contest dataset, $d = 480,189$ and $w = 17,770$.

Code 1 shows the compute steps of GNMF[1]. There are two updates for Matrix $\mathbf{H}$ and $\mathbf{W}$ during each iteration. The common matrix multiplication of $\mathbf{W} \times \mathbf{H}$ is not shared between the two compute steps (in Lines 7 and 8), because $\mathbf{H}$ is updated during the execution. Observe that the matrix chain multiplications $\mathbf{W}^T \times \mathbf{W} \times \mathbf{H}$ and $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$ involve more than one matrix multiplication. The order of execution on multiple matrix multiplications should be chosen carefully to avoid generating intermediate matrices of large sizes. The matrix metadata records several properties, e.g., the dimension, the sparsity (dense or sparse), and the storage format. From the metadata of the input matrices, it should be possible to infer the dimensions of intermediate matrices. For computing $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$, there are two possible execution orders, i.e., computing $\mathbf{W} \times \mathbf{H}$ first and producing an intermediate result with $480,189 \times 200 \times 17,770$ arithmetic multiplications; or computing $\mathbf{H} \times \mathbf{H}^T$ first and producing an intermediate result with $200 \times 17,770 \times 200$ arithmetic multiplications; assuming the Netflix contest dataset when p is set to 200. The two execution plans differ greatly in the dimensions of the intermediate matrices, and result in different computation costs. The plan generation becomes even more intricate when sparse matrices are involved. It is usually difficult to obtain accurate estimates on the sparsity of the computed intermediate matrices, which is directly related to the computation cost. Another common feature of matrix programs is the mix between element-wise operations and matrix-matrix multiplications. An eager plan generator that arranges a sequential execution incurs unnecessary memory overhead for intermediate matrices. An optimizer for matrix computation is needed to leverage features of matrix programs that reduce computation and memory overhead.

```
1  val p = 200 // number of topics
2  val V = loadMatrix("in/V") // read matrix
3  var W = RandomMatrix(V.nrows, p)
4  var H = RandomMatrix(p, V.ncols)
5  val max_iteration = 10
6  for (i <- 0 until max_iteration) {
7     H = H * (W.t %*% V) / (W.t %*% W %*% H)
8     W = W * (V %*% H.t) / (W %*% H %*% H.t)
9  }
10 W.saveAsTextFile("out/W")
11 H.saveAsTextFile("out/H")
```

Code 1: GNMF algorithm in Scala

In a distributed setup, the communication overhead may become a bottleneck in matrix computations. Load-balanced data partitioning schemes, e.g., hash-based schemes, where a hash function distributes rows evenly across the partitions, may not be efficient for matrix operations with data dependencies. Data dependencies between compute steps in a matrix program are prevalent, e.g., an update of Matrix $\mathbf{H}$ (Line 7) is fed to compute a new update of Matrix $\mathbf{W}$ (Line 8). The matrix multiplication $\mathbf{H} \times \mathbf{H}^T$ will requires re-partitioning if a hash-based scheme is used for $\mathbf{H}$. Thus, optimizing the data partitioning of input and intermediate matrices is also a critical step for efficiently executing matrix programs.

In this paper, we introduce MatFast, an in-memory distributed matrix computation processing system. MatFast has (1) a matrix program optimizer to identify and leverage special features of the input matrices to reduce computation cost and memory footprint, and (2) a matrix data partitioner to mitigate communication overhead. The matrix program optimizer uses a cost model and heuristic rules to dynamically generate an execution plan. MatFast uses a second cost model to partition the input and intermediate matrices to minimize communication overhead. To improve compute performance on local matrices, MatFast leverages a block-based strategy for efficient local matrix computations. MatFast is designed as a Spark library that uses Spark's standard dataflow operators.

The main contributions of this paper are as follows:

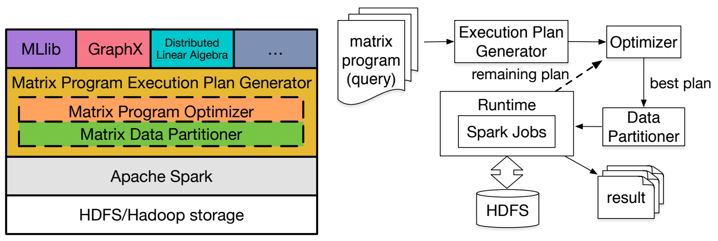
- We develop MatFast, a matrix computation system for efficiently processing and optimizing matrix programs in a distributed in-memory environment (Section II).
- We introduce a cost model to accurately estimate the sparsity of sparse matrix multiplications, and propose heuristic rules to rewrite special features of a matrix program for mitigating memory footprint (Section III).
- We introduce a second cost model to distribute the matrix data partitions among various workers for a communication-efficient execution (Section IV).
- We evaluate MatFast against state-of-the-art distributed matrix computation systems using real and synthetic datasets. Experimental results illustrate up to an order of magnitude enhancement in performance (Section V).

II. Preliminaries

Notation. We follow the convention that a bold, upper-case Roman letter, e.g., $\mathbf{A}$, denotes a matrix and regular Roman letter with subscripts, e.g., A_{ij} represent single elements in a matrix. A column vector is written in bold, lower-case Roman letter, e.g., $\mathbf{x}$. A scalar is written in lower-case Greek letter, e.g., β. A block matrix is written as $\mathbf{A}_{ij}$, where i and j are the row-block index and column-block index. (X_{ij}) represents a matrix with element X_{ij} at the i-th row and the j-th column.

Matrix Operators. MatFast supports a variety of unary and binary matrix operators. MatFast supports the unary matrix transpose $\mathbf{B} = \mathbf{A}^T$, $B_{ij} = A_{ji}, \forall i, j$. Binary operators includes matrix-scalar addition, $\mathbf{B} = \mathbf{A} + \beta$, $B_{ij} = A_{ij} + \beta$; matrix-scalar multiplication, $\mathbf{B} = \mathbf{A} * \beta$, $B_{ij} = \beta * A_{ij}$; matrix-matrix addition, matrix-matrix element-wise multiplication, matrix-matrix element-wise division, $\mathbf{C} = \mathbf{A} \star \mathbf{B}$, $C_{ij} = A_{ij} \star B_{ij}$, where $\star \in \{+, *, /\}$; and matrix-matrix multiplication, $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, $C_{ij} = \sum_k A_{ik} * B_{kj}$. In addition, MatFast supports the following matrix functions: $\texttt{abs}(\mathbf{A}) = (|A_{ij}|)$; $\texttt{max}(\mathbf{A}) = \max\{A_{ij}\}$; $\texttt{min}(\mathbf{A}) = \min\{A_{ij}\}$; $\texttt{rowSum}(\mathbf{A}) = \sum_j A_{ij}$; $\texttt{colSum}(\mathbf{A}) = \sum_i A_{ij}$; and $\texttt{pow}(\mathbf{A}, p) = \left(A_{ij}^p\right) \forall i, j$. By default, matrix transpose

[1]In the script, %*% denotes matrix-matrix multiplications, and $*$ (/) denotes element-wise multiplication (division) between two matrices, respectively. W.t denotes the transpose of Matrix W.



(a) Components of MATFAST (b) Architecture of MATFAST

Fig. 1: An Overview of MATFAST

has the highest precedence, and matrix-matrix multiplication has a higher precedence over element-wise operators.

A. Overview of Distributed Matrix Computation Processing

To facilitate matrix computation, we realize an execution plan generator for evaluating matrix expressions over in-memory matrix data. Given an analytic task, e.g., an ML algorithm, that involves multiple matrix expressions, these expressions are extracted and are optimized to generate a compute- and memory-efficient logical evaluation plan. Then, we develop a cost model to decide on how the input and intermediate matrix data are partitioned based on data dependencies. Finally, each worker adopts a block-based matrix storage to execute computations locally.

Refer to Figure 1 for illustration. MATFAST consists of three major components: a plan generator for executing matrix programs, a query optimizer, and a data partitioner. These components leverage rules to transform a matrix expression (that is extracted from a high-level application) to an optimized execution plan in a distributed environment. Figure 1b gives the workflow among the various components. For each matrix expression or query (a matrix expression is a query for MATFAST), the execution plan generator produces an initial query evaluation plan tree that is pipelined into the query optimizer to apply cost-based dynamic analysis and rule-based rewriting heuristics. The matrix data partitioner assigns partitioning schemes to input and intermediate matrices based on a cost model. For matrix expressions that involve sparse matrix multiplications, a globally optimal execution plan cannot be determined by a single pass on the plan tree due to inaccurate estimates on the computation cost of the intermediate matrices. MATFAST adopts a greedy approach to progressively generate an execution order for sparse matrix chain multiplications. The dashed arrow in Figure 1b refers to the dynamic optimizations of these cases.

III. EXECUTION PLAN GENERATION

We present how to generate a computation- and communication-efficient execution plan for a matrix expression. First, we present a sampling-based technique to estimate the computation cost for sparse matrix chain multiplications in a single statement. Next, we introduce rule-based heuristics to identify special features of a matrix expression for memory efficiency. Finally, we present a cost model to estimate the communication overhead for optimizing the data partitioning of individual input and intermediate matrices in the matrix expression.

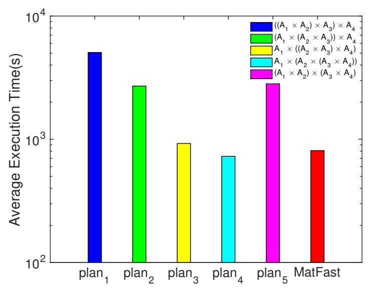


Fig. 2: Computation costs of different plans

A. Cost-based Dynamic Optimization

Matrix chain multiplication is commonly found in random walk [23] and matrix factorization [24] applications. We distinguish between dense and sparse matrix chain multiplications. For dense matrix chain multiplications, MATFAST exploits the classical dynamic programming approach [14] to determine the optimal order of the matrix pair multiplications. The cost of multiplying two dense matrices is defined as the number of arithmetic floating point multiplications. The computation cost of $\mathbf{A}_{m\times q} \times \mathbf{B}_{q\times n}$ can be estimated by mqn floating point multiplications. However, for sparse matrix chain multiplications, we cannot apply the same dynamic programming approach, because the computation cost of intermediate product matrices would be over-estimated. This cost depends not only on the dimensions of the input matrices but also on several other factors, e.g., matrix sparsity and the locations of non-zero entries. Figure 2 gives the average runtime of various plans for a sparse matrix chain multiplication of length 4. The cost varies significantly between the best and the worst plan. To better estimate this computation cost, various sparsity estimation methods, e.g., average-case estimation [22] and worst-case estimation [11], [32], can be used and are explained below.

Given a sparse matrix, the associated metadata contains the dimension and sparsity information, i.e., the number of rows m, columns n, and the sparsity ρ, where $\rho = N_{nz}/(mn)$, N_{nz} is the number of non-zero entries. For matrix-matrix multiplication $\mathbf{C}_{m\times n} = \mathbf{A}_{m\times q} \times \mathbf{B}_{q\times n}$, estimating the sparsity ρ_{c} of Matrix $\mathbf{C}$ is difficult, and is usually interpreted as the probability of non-zero entries in a matrix, under the uniform distribution assumption. Thus, the average- and worst-case estimations predict sparsity as $\rho_{\text{c}} = 1 - (1 - \rho_{\text{A}}\rho_{\text{B}})^q$, and $\rho_{\text{c}} = \min(1, \rho_{\text{A}} q) \times \min(1, \rho_{\text{B}} q)$, respectively.

For matrices derived from real-world applications, e.g., online social networks, non-zero entries usually follow non-uniform distributions. Average-case estimation works poorly for these matrices. The node degrees follow a power law distribution [25], where certain rows and columns contain substantially more non-zero entries than others. Worst-case estimation is pessimistic, and leaves little opportunity for optimization, i.e., it generates a sequential execution plan of the multiplication chain since the sparsity is estimated to

1 when $\rho_{\text{A}} q \geq 1$ and $\rho_{\text{B}} q \geq 1$. Average- and worst-case estimations are static because they predict sparsity without touching the underlying matrices.

Thus, cost estimation for sparse matrix chain multiplications should conservatively consider data skew. Matrix-matrix multiplication $\mathbf{A} \times \mathbf{B}$ can be interpreted as the summation of the vector outer products between corresponding columns from $\mathbf{A}$ and rows from $\mathbf{B}$, i.e.,

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_k \end{bmatrix} \times \begin{bmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \vdots \\ \mathbf{b}_k^T \end{bmatrix} = \sum_{i=1}^{k} \mathbf{a}_i \mathbf{b}_i^T.$$

This matrix multiplication rule also works for block partitioned matrices, where $\mathbf{A}$ and $\mathbf{B}$ are partitioned into compatible blocks, i.e., the number of columns in block $\mathbf{A}_{ik}$ equals the number of rows in block $\mathbf{B}_{kj}$. The outer-product perspective provides a different way to estimate the cost of sparse matrix multiplications. Intuitively, a larger product value of $\texttt{nnz}(\mathbf{a}_i) \times \texttt{nnz}(\mathbf{b}_i^T)$ leads to a denser multiplication matrix, where $\texttt{nnz}(\mathbf{A})$ denotes the number of non-zero entries in Matrix $\mathbf{A}$. However, it is unaffordable to calculate each $\texttt{nnz}(\mathbf{a}_i) \times \texttt{nnz}(\mathbf{b}_i^T)$ for large matrices with millions of rows or columns. The optimizer needs a sketch about the exact cost.

To obtain an accurate cost estimation of sparse matrix chain multiplications, MatFast adopts a sampling-based approach to sketch the positions of the non-zero entries. A good sampling method needs to capture the densest columns and rows. If the number of non-zero entries in a row (column) of a sparse matrix follows a power law distribution, and the rows (columns) are in the descending order with respect to the number of nonzero entries, then it is ideal to select the first few rows and columns for estimating the computation cost of multiplying the matrices. If no prior knowledge is available for the input, MatFast adopts a simple random-sampling method, e.g., systematic sampling [31], to estimate the computation cost of the multiplication. This cost estimation can be generalized to block partitioned sparse matrix multiplication as follows,

$$C_{comp}(\mathbf{A} \times \mathbf{B}) = \max_{k \in \mathcal{S}} \{c_k * r_k\},$$
$$\text{where } c_k = \sum_i \texttt{nnz}(\mathbf{A}_{ik}), r_k = \sum_j \texttt{nnz}(\mathbf{B}_{kj}),$$

where $C_{comp}(\mathbf{X})$ denotes the computation cost of calculating Matrix $\mathbf{X}$, and $\mathcal{S}$ is the set of the sampled column (row) block indices, and c_k (r_k) is the number of non-zero entries in the k-th column (row) block. The maximum operator is used because a larger value of $c_k * r_k$ indicates a denser product matrix.

Analysis of Cost Estimation with Sampling. If the distribution of the non-zero entries is provided by a user, MatFast samples rows (columns) according to the distribution. If MatFast samples input matrices with a random sampling method, the probability of accurately estimating the cost can be modeled as follows. Suppose there are n columns in Matrix $\mathbf{A}$, n rows in Matrix $\mathbf{B}$, and w column-row pairs, whose products achieve the maximum product. The probability that the maximum product is chosen in s samples is,

$$P = 1 - \binom{n-w}{s} \binom{n}{s}^{-1}.$$

By sampling s row-column pairs, there are totally $\binom{n}{s}$ possible combinations. The chance that the maximum pair is not chosen among w pairs is $\binom{n-w}{s}\binom{n}{s}^{-1}$. Thus, P can be computed by the complementary event. Similarly, for a block partitioned matrix, this probability is as follows:

$$\hat{P} = 1 - \binom{\hat{n}-\hat{w}}{s} \binom{\hat{n}}{s}^{-1},$$

where $\hat{n}$ is the number of column (row) blocks of matrix $\mathbf{A}$ ($\mathbf{B}$), $\hat{n} = n/\ell$ (ℓ is the block size), $\hat{w}$ is the number of blocks that achieves the maximum product value. In practice, $\hat{P} \geq P$ and the probability of accurate cost estimation is improved.

Running Example. Given a sparse matrix chain multiplication $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \mathbf{A}_4$, MatFast dynamically generates an execution plan. Initially, the sampling index set $\mathcal{S}$ is determined by a random sampling method. The number of non-zero entries are collected for the sampled rows and columns. Next, the costs for pair-wise adjacent matrix multiplications are computed, i.e., $c_1 = C_{comp}(\mathbf{A}_1 \times \mathbf{A}_2)$, and similarly for c_2 and c_3. Say, c_2 is the cheapest. The multiplication chain is computed a step further and is reduced to $\mathbf{A}_1 \times \mathbf{M}_{23} \times \mathbf{A}_4$, where $\mathbf{M}_{ij}$ is the intermediate product matrix of $\mathbf{A}_i$ and $\mathbf{A}_j$. Then, the sampling is conducted again on $\mathbf{M}_{ij}$. Notice that the sampled statistics can be reused for the existing matrices, e.g., $\mathbf{A}_1$ and $\mathbf{A}_4$. The sampling-based cost estimation repeats until the chain is reduced to a single matrix.

B. Rule-based Heuristics

Matrix expressions have various features that may induce heavy memory footprints. We identify the following features: (1) low-rank matrix updates, (2) chains of multiple element-wise matrix operators, and (3) loop structures that reflect iterative executions. MatFast handles these features by using heuristics to generate a memory-efficient execution plan.

Identifying and Preserving Low-rank Matrix Updates. Low-rank matrix updates are widely used in ML models due to the popularity of latent variables [19]. For example, the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [26] and its variant limited memory BFGS (ℓ-BFGS) are widely used quasi-Newton methods for solving unconstrained nonlinear optimization problems. Low-rank matrix update is a critical step of BFGS and is stated as follows:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{y}_k \times \mathbf{y}_k^T}{\mathbf{y}_k^T \times \mathbf{s}_k} - \frac{\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k}{\mathbf{s}_k^T \times \mathbf{B}_k \times \mathbf{s}_k},$$

where $\mathbf{B}_k$ is the approximate Hessian matrix, $\mathbf{s}_k$ is the line search step, and $\mathbf{y}_k$ is the difference of the gradient. $\mathbf{y}_k^T \times \mathbf{s}_k$ and $\mathbf{s}_k^T \times \mathbf{B}_k \times \mathbf{s}_k$ are two scalars that can be easily computed and shared among workers. Notice that there are two low-rank (rank-1) matrix updates in each iteration, i.e., $\mathbf{y}_k \times \mathbf{y}_k^T$ and $\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k$. An ignorant query optimizer generates a sequential execution plan for each intermediate matrix, i.e., both low-rank update matrices have to be explicitly computed and materialized during the execution. However, multiplying

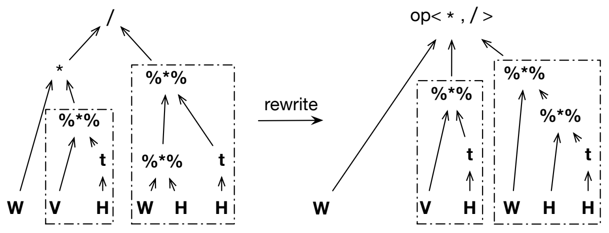


Fig. 3: Element-wise operators folding

low-rank matrices usually produces dense matrices of very high dimensions, and incurs heavy memory overhead.

Given a matrix expression, MATFAST analyzes the dimensions of the input matrices, and identifies low-rank matrix updates. It defers the evaluation of low-rank matrix updates to reduce the memory footprint. Low-rank matrix updates usually involve matrix components of low dimensions. Thus, storing and transmitting the low-rank matrix components are more efficient than materializing the matrix product. For example, matrix $\mathbf{y}_k$ and $\mathbf{B}_k \times \mathbf{s}_k$ are two vectors but their corresponding rank-1 updates $\mathbf{y}_k \times \mathbf{y}_k^T$ and $\mathbf{B}_k \times \mathbf{s}_k \times \mathbf{s}_k^T \times \mathbf{B}_k$ are dense. To evaluate the updated Hessian matrix $\mathbf{B}_{k+1}$, first, we broadcast vector $\mathbf{y}_k$ and $\mathbf{B}_k \times \mathbf{s}_k$ to all workers. Then, each element of the low-rank matrix multiplication is computed from the vectors on the fly without storing the matrix product explicitly.

Folding of Matrix Operators. Matrix element-wise operations are prevalent in ML algorithms. For GNMF (Code 1), the algorithm updates Matrix $\mathbf{W}$ with element-wise multiplications and divisions, i.e., $\mathbf{W} = \mathbf{W} * (\mathbf{V} \times \mathbf{H}^T) / (\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T)$. These expressions are generalized as $\mathbf{A}_1 \star \mathbf{A}_2 \star \cdots \star \mathbf{A}_k$, where all $\mathbf{A}_i$'s have the same dimension and $\star$'s are element-wise operators, i.e., $\star \in \{+, *, /\}$. For the update of $\mathbf{W}$, $\mathbf{A}_1 = \mathbf{W}, \mathbf{A}_2 = \mathbf{V} \times \mathbf{H}^T$, and $\mathbf{A}_3 = \mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$. To generate an execution plan for this expression, a naive query optimizer generates a "*left-deep tree*" plan. The left part of Figure 3 shows the left-deep tree plan for updating Matrix $\mathbf{W}$, where the dashed rectangles represent the subtrees for $\mathbf{A}_2$ and $\mathbf{A}_3$. The problem is that the inner nodes of the left-deep tree plan must be materialized, e.g., $\mathbf{W} * (\mathbf{V} \times \mathbf{H}^T)$, before the element-wise division. A left-deep tree plan induces multiple compute steps and memory overhead for storing the intermediate matrices.

Given a matrix expression, MATFAST identifies element-wise matrix operations and then organizes them in a "*bushy tree*" execution plan. A bushy tree plan benefits from circumventing materializing the intermediate matrices. The right part of Figure 3 illustrates the bushy tree plan for updating Matrix $\mathbf{W}$. To facilitate low-level executions, MATFAST's optimizer generates a tree node of a new compound operator in the form "`op<binop_list>`". The compound operator records each element-wise operator and the corresponding input matrices. For example, "`op<`$\star$`,/>`" in Figure 3 encodes the chain of matrix element-wise multiplication and division operations among $\mathbf{W}$, $\mathbf{A}_2$, and $\mathbf{A}_3$. Notice that the dashed rectangle also gives an optimized plan for intermediate Matrix $\mathbf{A}_3$. The optimization reflects the execution plan of $\mathbf{W} \times \mathbf{H} \times \mathbf{H}^T$.

Eliminating Common and Invariant Expressions. MATFAST identifies and eliminates recurring common subexpressions (CSE) [8]. It recognizes loop-constant subexpressions, moves them out of the loop, and saves them for reuse. Thus, redundant computation and memory footprint are mitigated.

C. Plan Generation with Matrix Data Partitioning

When an optimized plan is distributed among a set of workers, its execution may suffer from heavy communication overhead due to inconsistent data partitioning schemes between dependent matrices. For GNMF (Code 1), the computation of $\mathbf{H} \times \mathbf{H}^T$ relies on the previous step. If $\mathbf{H}$ is partitioned in the default hash-based scheme, then we need to repartition $\mathbf{H}$ before the actual execution, and hence inducing extra communication cost. Thus, choosing a consistent matrix partitioning scheme for an intermediate matrix is essential to reduce communication overhead. In this section, we introduce MATFAST's matrix data partitioning schemes. Then, we present a cost model to efficiently partition matrix data.

Partitioning Schemes. MATFAST supports the following three matrix data partitioning schemes: *Row* ("*r*"), *Column* ("*c*"), and *Block-Cyclic* ("*b-c*"). Moreover, the *Broadcast* ("*b*") scheme is supported for sharing a matrix of low dimensions or a single vector. The Row and Column schemes place all the elements in the same row and column on a worker, respectively. The Block-Cyclic scheme partitions a matrix into many more blocks than the number of available workers, and assigns blocks to workers in a round-robin manner so that each worker receives several non-adjacent blocks. Different schemes introduce different communication costs for various matrix operators. They should be assigned to input matrices on their own merits. For example, for matrix-matrix multiplications with cross product plans [18], the Block-Cyclic scheme incurs extra shuffle cost to aggregate rows (columns) together. Therefore, we introduce a cost model to evaluate the communication costs of different partitioning schemes for the various matrix operators.

Cost Model for Communication. The communication cost incurred by a matrix expression can be modeled by,

$$C_{comm}(expr_i) = \sum_{j=1}^{m} C_{comm}(op_j, s_{j_{i1}}[, s_{j_{i2}}], s_{j_o}),$$

where matrix expression $expr_i$ contains j matrix operators, and each operator takes some inputs in schemes $s_{j_{i1}}[, s_{j_{i2}}]$, and produces an output in scheme s_{j_o}. $[, s_{j_{i2}}]$ represents an optional argument as we support both unary and binary operators. For the unary operator op, $C_{comm}(op, s_i, s_o)$ is characterized by the input matrix and partitioning schemes in Table I, where $\mathbf{A}$ is the input matrix, s_i and s_o are the partitioning schemes of the input and the output, respectively, and N is the number of the workers in the cluster. $|\mathbf{A}|$ refers to the size of Matrix $\mathbf{A}$, i.e., $|\mathbf{A}| = mn$ if $\mathbf{A}$ is an m-by-n dense matrix; and it means $\texttt{nnz}(\mathbf{A})$ if $\mathbf{A}$ is sparse. If the input matrix is partitioned in the Row scheme, then the transposed matrix is naturally partitioned in the Column scheme. Therefore, no communication cost is introduced. However, if the input matrix

$s_o \setminus s_i$	r	c	b	b-c
r	$\|\mathbf{A}\|$	0	0	$\|\mathbf{A}\|$
c	0	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$
b	$N\|\mathbf{A}\|$	$N\|\mathbf{A}\|$	0	$N\|\mathbf{A}\|$
b-c	$\|\mathbf{A}\|$	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$

TABLE I: Communication cost of matrix transpose

$s_o \setminus s_i$	r	c	b	b-c
r	0	$\|\mathbf{A}\|$	0	$\|\mathbf{A}\|$
c	$\|\mathbf{A}\|$	0	0	$\|\mathbf{A}\|$
b	$N\|\mathbf{A}\|$	$N\|\mathbf{A}\|$	0	$N\|\mathbf{A}\|$
b-c	$\|\mathbf{A}\|$	$\|\mathbf{A}\|$	0	0

TABLE II: Communication cost of matrix-scalar operators

is partitioned in the Row scheme and the output matrix is also required to be partitioned in the Row scheme, then the matrix data must be shuffled to satisfy the requirement. This results in shuffling the whole matrix. Similarly, for matrix-scalar operators (e.g., multiplying a matrix by a constant), Table II gives the communication costs for the various schemes. If the input and the output are partitioned in the same scheme, then there is no communication. Notice that no communication is incurred if the inputs are partitioned in the Broadcast scheme.

Let $C_{comm}(op, s_{i1}, s_{i2}, s_o)$ be the cost function for a matrix element-wise operator that is illustrated in Table III. From the table, the matrix element-wise operators introduce no communication overhead if both input matrices are partitioned (1) in the same scheme as the output, (2) at least one of the inputs is partitioned in the Broadcast scheme and the other one has the same scheme as the output.

Matrix-matrix multiplication is a bit more complicated. We do not use the Block-Cyclic scheme as it incurs more overhead than the other schemes. The cost function is given in Table IV. Notice that the cells with 0's in the table indicate no cost, e.g., when the inputs are partitioned in the Row scheme, the Broadcast scheme, and the output is in the Row scheme.

With the cost functions introduced above, MATFAST assigns the partitioning schemes having minimum costs to the input and intermediate matrices, i.e.,

$$s_{i1(i2)} \leftarrow \underset{s_{i1(i2)}}{\arg\min}\, C_{comm}(op, s_{i1}[, s_{i2}], s_o).$$

MATFAST optimizes the communication cost for a single operator, and assigns the associated scheme to the input. For a matrix expression consisting of several operators, the entire expression is greedily optimized by tuning each operator.

Algorithm for Plan Generation and Partitioning Scheme Assignment. Algorithm 1 describes plan generation in MATFAST. The input is a matrix program P, and the output is an optimized execution plan tree T with an optimized partitioning scheme at each node. MATFAST applies the rule-based heuristics to each expression of P. If the expression contains no matrix operator, then the variable is parsed and associated with the corresponding metadata (Line 30), e.g., loading matrix data or storing results to HDFS. If an expression contains matrix chain multiplications, the execution plan is determined by the matrix types (Lines 19-26), i.e., a classical dynamic programming approach is invoked for dense matrix chains; otherwise, dynamic cost-estimation is triggered

s_o	(s_{i1}, s_{i2}) Communication Cost		
r	{(r, r), (r, b), (b, r), (b, b)} 0		else $\|\mathbf{A}\|$
c	{(c, c), (c, b), (b, c), (b, b)} 0		else $\|\mathbf{A}\|$
b	(b, b) 0	$s_{i1} = s_{i2}$ or {(b, *), (*, b)} $N\|\mathbf{A}\|$	else $(N+1)\|\mathbf{A}\|$
b-c	{(b-c, b-c), (b, b), (b-c, b), (b, b-c)} 0	{(b-c, *), (*, b-c)} $\|\mathbf{A}\|$	else $2\|\mathbf{A}\|$

TABLE III: Communication cost of element-wise operators

for sparse matrix chain multiplications (Line 24). Then, the optimized expression is inserted into the plan tree T (Line 27). Finally, Procedure ASSIGNPARITIONSCHEME (Algorithm 2) assigns partitioning schemes to matrices based on the cost model. The scheme assignment starts from the root of the plan tree T. For the root node, the scheme is determined by the nature of the output, i.e., recurring in a loop, participating in matrix-matrix multiplications, or only involving in element-wise operations. For an internal node, the Broadcast scheme is assigned if it consists of low-rank matrix updates. Otherwise, an internal node is assigned with a scheme such that it introduces minimum communication cost (Lines 17 and 20). After scheme assignment, the execution is organized into several *stages*, where the operations are packed together such that no communication is introduced in the same stage.

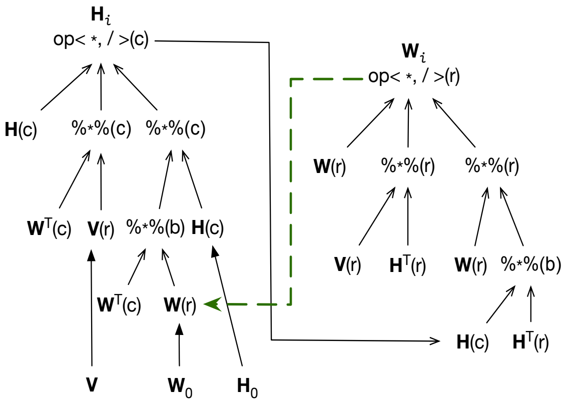
GNMF Running Example. The optimized execution plan tree is given in Figure 4a. Procedure ASSIGNPARTITIONSCHEME traverses from the root node to all leaf nodes, and assigns a partitioning scheme to each input and intermediate matrix. The left and right subtrees are the execution plans for updating Matrices $\mathbf{H}$, and $\mathbf{W}$, respectively. The dashed arrows indicate loop execution. The root node $\mathbf{W}_i$ participates in matrix-matrix multiplication and loop execution. The number of rows in $\mathbf{W}_i$ is significantly bigger than that of columns. Thus, the Row scheme leads to a more balanced data distribution. The cost model for the element-wise operator aids in assigning the Row schemes to $\mathbf{W}_i$'s child nodes. To determine the partitioning schemes for $\mathbf{V}$ and $\mathbf{H}^T$, the procedure checks the cost model table for matrix-matrix multiplication. The dimension of $\mathbf{V}$ is larger than that of the product $\mathbf{V} \times \mathbf{H}^T$. Thus, partitioning both matrices in the Row scheme is cheaper than other strategies, i.e., $17,770 \times 200 \times N$. Matrix $\mathbf{H} \times \mathbf{H}^T$ is partitioned in the Broadcast scheme due to its tiny dimension (200×200). Similarly, $\mathbf{H}$ uses the Column scheme and $\mathbf{H}^T$ uses the Row scheme with no cost. $\mathbf{H}_i$'s subtree is processed similarly. Figure 4b gives the physical execution in the cluster. The dashed boxes indicate the execution stages.

IV. LOCAL EXECUTION AND SYSTEM IMPLEMENTATION

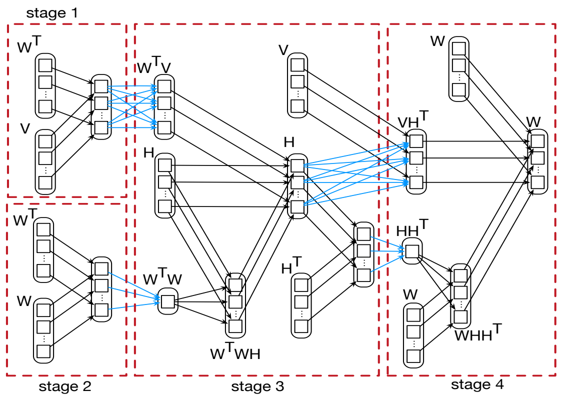
Once the query execution with matrix partitioning schemes is generated, each compute node locally performs matrix computations. We use block matrices as a basic unit for manipulation to store matrix data in the distributed memory. We discuss briefly the system implementation on top of Spark.

	(s_{i1}, s_{i2})								
s_o	(r, r)	(r, c)	(r, b)	(c, r)	(c, c)	(c, b)	(b, r)	(b, c)	(b, b)
r	$N\lvert\mathbf{B}\rvert$	$N\lvert\mathbf{B}\rvert$	0	$N\lvert\mathbf{AB}\rvert$	$N(\lvert\mathbf{A}\rvert + \lvert\mathbf{B}\rvert)$	$\lvert\mathbf{A}\rvert$	$N\lvert\mathbf{AB}\rvert$	$\lvert\mathbf{AB}\rvert$	0
c	$N(\lvert\mathbf{A}\rvert + \lvert\mathbf{B}\rvert)$	$N\lvert\mathbf{A}\rvert$	$\lvert\mathbf{AB}\rvert$	$N\lvert\mathbf{AB}\rvert$	$N\lvert\mathbf{A}\rvert$	$N\lvert\mathbf{AB}\rvert$	$\lvert\mathbf{B}\rvert$	0	0
b	$\min\{N\lvert\mathbf{A}\rvert + C(b,r), N\lvert\mathbf{B}\rvert + C(r,b)\}$	$N\min\{\lvert\mathbf{A}\rvert, \lvert\mathbf{B}\rvert\} + N\lvert\mathbf{AB}\rvert$	$N\lvert\mathbf{AB}\rvert$	$2(N-1)\lvert\mathbf{AB}\rvert$	$\min\{N\lvert\mathbf{A}\rvert + C(b,c), N\lvert\mathbf{B}\rvert + C(c,b)\}$	$\min\{\lvert\mathbf{A}\rvert + N\lvert\mathbf{AB}\rvert, 2(N-1)\lvert\mathbf{AB}\rvert\}$	$\min\{\lvert\mathbf{B}\rvert + N\lvert\mathbf{AB}\rvert, 2(N-1)\lvert\mathbf{AB}\rvert\}$	$N\lvert\mathbf{AB}\rvert$	0

TABLE IV: Communication cost of matrix-matrix multiplications. $C(s_{i1}, s_{i2})$ is the cost when the 2 matrices are partitioned in schemes s_{i1} and s_{i2}.



(a) Optimized query execution plan of GNMF



(b) Physical execution plan of GNMF

Fig. 4: Execution plan with matrix data partitioning scheme of GNMF. In the physical plan, the blue lines denote the data shuffle among different data partitions. The dashed red rectangles denote different stages for the execution on Spark.

A. Physical Storage of a Local Matrix

To better utilize spatial locality of nearby entries in a matrix, we use block matrices to store matrix data in the distributed memory. A matrix block is the basic unit for storage and computation. Figure 5 illustrates that Matrix $\mathbf{A}$ is partitioned into blocks of size 3×3, where each block is stored as a local matrix. For the sake of simplicity, we only consider square blocks. Figure 5 illustrates an example storage layout of a block matrix. The block size may not be applicable to the last row (column) block, e.g., the row block with ID = 2. To fully exploit the compute power of CPU cores, MATFAST assigns each core 4 matrix blocks to each worker, i.e., $MN = 4WP\ell^2$, where M and N are the dimensions of the matrix, W is the number of workers, P is the number of cores per worker, and ℓ is the block size. To avoid performance degeneration, MATFAST limits the smallest block size to be 1000, i.e., $\ell = \max\left\{\sqrt{\frac{MN}{4WP}}, 1000\right\}$.

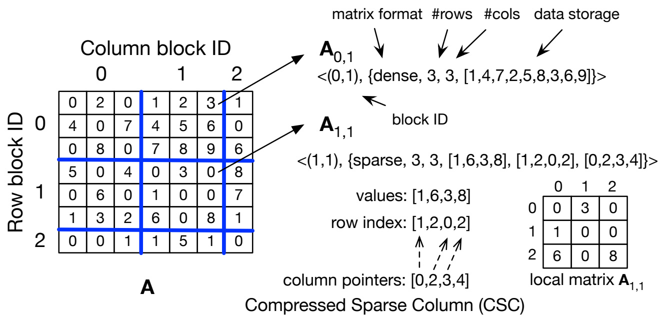

matrix format
#rows
#cols
data storage
Column block ID
0
1
2
Row block ID
0
1
2
$\mathbf{A}_{0,1}$
<(0,1), {dense, 3, 3, [1,4,7,2,5,8,3,6,9]}>
block ID
$\mathbf{A}_{1,1}$
<(1,1), {sparse, 3, 3, [1,6,3,8], [1,2,0,2], [0,2,3,4]}>
values: [1,6,3,8]
row index: [1,2,0,2]
column pointers: [0,2,3,4]
Compressed Sparse Column (CSC)
local matrix $\mathbf{A}_{1,1}$
A

Fig. 5: Block matrix storage

Each local matrix block consists of two components; a block ID and matrix data. A block ID is an ordered pair, i.e., (row-block-ID, column-block-ID). The matrix data field is a quadruple, ⟨matrix format, number of rows, number of columns, data storage⟩. A local matrix block supports dense and sparse matrix storage formats. For the dense format, an array of double precision floating point numbers stores all block entries. For the sparse format, the non-zero entries are stored in Compressed Sparse Column (CSC), and Compressed Sparse Row (CSR) format [15]. The compressed format, say CSC, requires three arrays to store all the data. Array *values* stores all the non-zero entries, and Array *row index* records the row index for the corresponding entry. Array *column pointers* keeps the starting position of each column, and the last entry records the total number of non-zero entries. Figure 5 illustrates the dense representation of Block $\mathbf{A}_{0,1}$, and the compressed representation (CSC) of Block $\mathbf{A}_{1,1}$. An $m \times n$ sparse matrix in CSC format requires $(2N_{nz}+n+1)\times 8$ bytes.

For dense matrices, we leverage high-performance dense matrix kernels to conduct matrix operations locally, e.g., the LAPACK kernel. Unlike existing systems, e.g., MLlib [5], MATFAST operates on sparse matrices in compressed formats directly, without converting to their dense counterparts. Performing local matrix computations in compressed format mitigates the memory footprint for operations among large sparse matrices. This is confirmed by our experiments in the PageRank and sparse matrix chain multiplications case studies.

B. System Design and Implementation

MATFAST is implemented as a library in Spark, and provides Scala API for conducting distributed matrix computa-

Algorithm 1: Execution plan generation

```
Input: Matrix program $P$
Output: Execution plan tree $T$
$T \leftarrow \emptyset$
$L \leftarrow P.getExprList()$ // expressions in sequential order
for $expr$ in $L$ do
    $currExpr \leftarrow expr$
    if $expr.containsMatrixOperator()$ then
        // low-rank matrix preservation
        if $expr.containsLowRankMatrix()$ then
            $currExpr \leftarrow preserveLowRank(currExpr)$
        end
        // operator folding
        if $expr.elementOperator() > 1$ then
            $currExpr \leftarrow createCompound(currExpr)$
        end
        // loop invariant extraction
        if $expr.isLoop()$ && $expr.containsConst()$ then
            $currExpr \leftarrow extractConst(currExpr)$
        end
        // matrix chain multiplications
        if $expr.containsMatrixChainMult()$ then
            if $isDenseChain(expr)$ then
                $currExpr \leftarrow densePlan(currExpr)$
            end
            else
                $currExpr \leftarrow sparsePlan(currExpr)$
            end
        end
        $T.add(currExpr)$
    end
    else
        load data into memory and extract metadata
    end
end
$assignPartitionScheme(T, null)$
return $T$
```

Algorithm 2: ASSIGNPARTITIONSCHEME(T, q)

```
// Plan tree $T$, output matrix partitioning scheme $q$
// for root node
if $q = null$ then
    if $T.type =$ 'compound' && (($isInLoop(T)$ && $!inMatMult(T)$) || $!isInLoop(T)$) then
        $T.scheme \leftarrow$ 'b-c'
    end
    else
        Choose $p \in \{$'r', 'c'$\}$ based on the metadata of $T$, and $T.scheme \leftarrow p$
    end
end
// for an internal node
else if $T.scheme = null$ then
    if $isLowRankMult(T)$ || $isTinySize(T)$ then
        $T.scheme \leftarrow$ 'b'
    end
    if $op$ is unary then
        $T.scheme \leftarrow \arg\min_{s_i} C_{comm}(op, s_i, q)$
    end
    else
        $T.scheme \leftarrow \arg\min_{s_{i1} \text{ or } s_{i2}} C_{comm}(op, s_{i1}, s_{i2}, q)$, with respect to the position of $T$
    end
end
for $R$ in $T.children$ do
    $assignPartitionScheme(R, T.scheme)$
end
```

tions. It uses RDD (Resilient Distributed Datasets) [34] to represent matrix blocks. A driver program orchestrates the executions of the various workers in the cluster. The dimension and sparsity statistics are computed and maintained at the driver program when a matrix is loaded into memory. The optimized execution plan and data partitioning schemes are generated at the driver program as well. The matrix operators are realized via RDD's transformation operations, e.g., map, flatMap, zipPartitions, and reduceByKey. Each local matrix has a block ID, and is stored as either a DenseMatrix or a SparseMatrix. Local matrix operations are optimized by the LAPACK kernel when conducting dense matrix computations. Sparse matrix computations, e.g., multiplication, are conducted in compressed format. The RDD Partitioner class is extended with the four Row, Column, Broadcast, and Block-Cyclic partitioning schemes for distributed matrices. MATFAST utilizes the caching mechanism of Spark to buffer a computed matrix when it repeatedly appears in the execution plan. Spark's fault tolerance mechanism applies naturally to MATFAST. The Spark cluster is managed by YARN, and a failure in the master node is detected and managed by ZooKeeper. In the case of master node failure, the lost master node is evicted and a standby node is chosen to recover the master. An open-source version of MATFAST is available at https://github.com/yuyongyang800/SparkDistributedMatrix.

V. CASE STUDIES AND EXPERIMENTS

We study the performance of the optimized execution plans by performing matrix operations from various ML applications. The performance is measured by the average execution time and communication (shuffle) cost. The experiments are conducted on two clusters: (1) a 4-node cluster, where one node has an Intel Xeon(R) E5-2690 CPU, 128GB memory, and a 2TB disk; the other 3, each has an Intel Xeon(R) E5-2640 CPU, 64GB memory, and a 2TB disk. The maximal memory allocation for JVM is 48GB, (2) Hathi cluster has 6 Dell compute nodes. Each has 2 8-core Intel E5-2650v2 CPUs, 32 GB memory, and 48TB of local storage. Spark 1.5.2 runs on YARN with the default configuration.

Case Studies. We conduct case studies on a series of ML models and matrix computations with special features on different datasets. These are PageRank, GNMF, BFGS, sparse matrix chain multiplications, and a biological data analysis.

Datasets. Our experiments are performed on both real-world and synthetic datasets. The six real-world datasets are: soc-pokec[2], cit-Patents[2], LiveJournal[2], Twitter2010[3], Netflix [9], and 1000 Genomes Project sample[4]. The synthetic datasets are generated by a sparse matrix generator by varying

[2]https://snap.stanford.edu/data/

[3]http://law.di.unimi.it/webdata/twitter-2010/

[4]http://www.1000genomes.org/

the dimensions, sparsity, skew type, and skew. The skew controls whether non-zero entries distribute in a row- or column-major way. To generate an $m \times n$ matrix with sparsity ρ and skew s in the column-major fashion, the generator produces $mn\rho$ values in a given range. The $mn\rho(1-s)^{j-1}$'s remaining elements are assigned randomly to the j-th column, until all the $mn\rho$ values are assigned. The PageRank experiments on soc-pokec, cit-Patents, LiveJournal, and synthetic sparse matrix chain multiplications are performed on Hathi. The rest are conducted on the first cluster as larger datasets require more memory on each worker node.

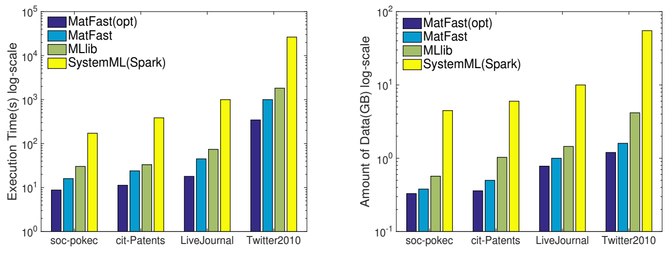
Baseline Comparison. Various matrix computation platforms have been recently proposed on Spark. In particular, we compare MATFAST with MLlib [5] and SystemML (https://github.com/apache/incubator-systemml,0.9)[18] in Spark batch mode. To validate the proposed optimizations, the results are presented in 2 modes, MatFast and MatFast(opt), where the optimizations for special matrix program features and data partitioning schemes are turned off in MatFast. MatFast partitions matrix data with the default hash partitioner in Spark. Moreover, the open-source library ScaLAPACK [7] and the array-based database SciDB [12] are used for performance evaluation.

Graph	#nodes	#edges
soc-pokec	1,632,803	30,622,564
cit-Patents	3,774,768	16,518,978
LiveJournal	4,847,571	68,993,773
Twitter2010	41,652,230	1,468,365,182

TABLE V: Statistics of the social network datasets

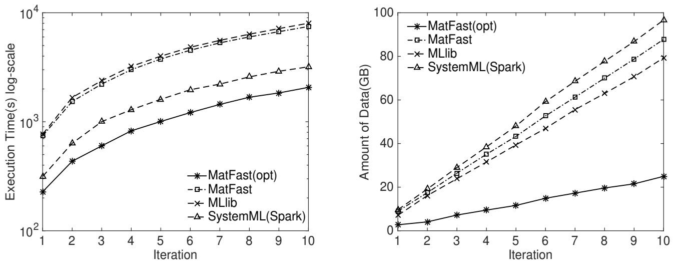
A. Case Studies

PageRank. The most expensive compute step of PageRank [23] is updating the vector, i.e., $\mathbf{x}_{k+1} = \alpha\mathbf{P}\mathbf{x}_k + (1-\alpha)\mathbf{v}$, where α is a constant scalar, $\mathbf{P}$ the stochastic link matrix, $\mathbf{v}$ the constant restart vector. The computation can be improved by eliminating constant sub-expressions from the loop, e.g., $\alpha\mathbf{P}$ and $(1-\alpha)\mathbf{v}$. Matrix $\mathbf{P}$ is partitioned in the Row scheme and Vector $\mathbf{x}$ and $\mathbf{v}$ are partitioned in the Broadcast scheme (from Table IV) because $\mathbf{x}$ and $\mathbf{v}$ are matrices of small sizes (two vectors). Table V lists all the statistics of the social network datasets used for the PageRank computation. All the graphs are sparse, and both MatFast(opt) and MatFast compute sparse matrix multiplications in compressed format. Figure 6a gives the average execution time for one iteration of PageRank on various social graph datasets. MATFAST consistently performs the best. MatFast(opt) outperforms MatFast when rule-based heuristics are turned on and optimized matrix data partitioning schemes are adopted. For Dataset Twitter2010, the average execution time per iteration in MatFast(opt) is about 340s, while it needs more than 1000s for MatFast, 1800s for MLlib and 26000s for SystemML in Spark mode. MatFast(opt) caches the invariant matrices in the distributed memory without repetitive computations. Figure 6b shows that MatFast(opt) incurs lowest shuffle costs during each iteration. MatFast(opt) outperforms the MLlib and SystemML due to the following reasons:



(a) Execution time (b) Communication cost

Fig. 6: PageRank on different real-world datasets



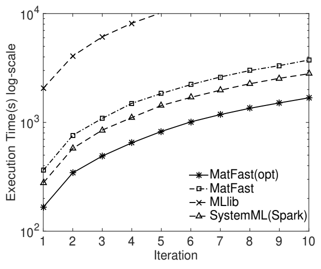
(a) Execution time (b) Communication cost

Fig. 7: GNMF on the Netflix dataset

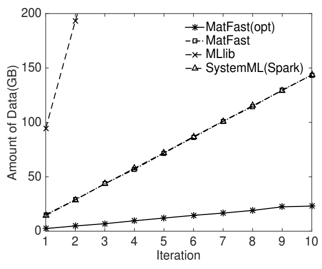
(1) it identifies and extracts the loop invariant expressions, and caches the invariants without recomputing, and (2) it broadcasts the updated PageRank vector based on the cost-efficient partitioning scheme.

GNMF. The Netflix dataset consists of 100,480,507 ratings on 17,770 movies from 480,189 customers. In the experiment, the number of topics p is set to 200. The performance of GNMF is optimized by folding element-wise matrix operators and choosing a cost-efficient order for matrix chain multiplications. Figure 7a gives the accumulated execution time for the Netflix dataset over different systems. MatFast(opt) consistently performs the best, followed by SystemML and MLlib. The accumulated execution time of MatFast is very close to that of MLlib. When all the optimization strategies are turned off, MatFast computes a matrix chain multiplication in the sequential order that is exactly the same order adopted by MLlib. The slight performance improvement is due to MatFast using replication-based matrix multiplications for tiny matrices and cross-product based matrix multiplications for other input matrix types. Both MatFast(opt) and SystemML optimize the dense matrix chain multiplications. MatFast(opt)'s extra speedup is due to its optimized matrix data partitioning schemes.

Figure 7b gives the accumulated communication costs for the Netflix dataset over various systems. MatFast(opt) has minimal communication overhead due to its effective data partitioning schemes. SystemML does not capture data dependencies among different matrix operators. Thus, it incurs high communication overhead. MLlib fails to identify a good execution plan for matrix chain multiplications, and lacks efficient data partitioning schemes for matrix data. When the data partitioning scheme assignment is turned off for MatFast, the communication overhead is similar to that of



(a) Execution time



(b) Communication cost

Fig. 8: BFGS on the Netflix dataset

SystemML. The reason is that MatFast adopts similar matrix multiplications strategies as SystemML.

BFGS. BFGS is widely used for solving unconstrained nonlinear optimization problems. For the Netflix problem, to obtain a good approximation of $\mathbf{V} \approx \mathbf{W} \times \mathbf{H}$, we can define an objective function as,

$$f(\mathbf{W}, \mathbf{H}) = ||\mathbf{V} - \mathbf{W} \times \mathbf{H}||_F^2 + ||\mathbf{W}||_F^2 + ||\mathbf{H}||_F^2,$$

where $||\mathbf{A}||_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2$. The gradient of $f(\mathbf{W}, \mathbf{H})$ is computed by the partial derivatives w.r.t. $\mathbf{W}$ and $\mathbf{H}$, i.e.,

$$\frac{\partial}{\partial \mathbf{W}} f(\mathbf{W}, \mathbf{H}) = 2\mathbf{W} - 2(\mathbf{V} - \mathbf{W} \times \mathbf{H}) \times \mathbf{H}^T,$$

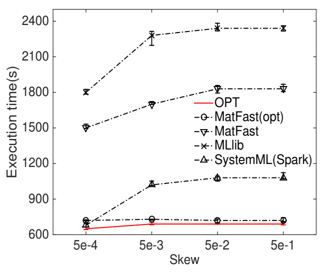
$$\frac{\partial}{\partial \mathbf{H}} f(\mathbf{W}, \mathbf{H}) = 2\mathbf{H} - 2\mathbf{W}^T \times (\mathbf{V} - \mathbf{W} \times \mathbf{H}).$$

A complete gradient for all the variables is derived by concatenating the two vectorized partial derivatives. We implement the key update computation for BFGS on the Netflix dataset.

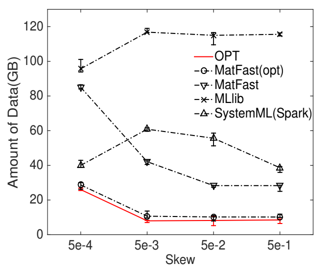
Figure 8a shows the accumulated execution time for BFGS update on the various systems. MatFast(opt) spends about 168s for each iteration using the low-rank matrix update heuristic. By turning off the optimization, MatFast spends about 372s for each iteration by storing the intermediate product matrices. SystemML performs slightly better than MatFast, and spends about 292s for each iteration. MLlib performs the worst and spends more than 2000s for a single iteration. The reason is that MLlib uses an inefficient strategy for matrix multiplications by duplicating copies of the input matrices. Figure 8b gives the communication overhead for each system. MatFast(opt) shuffles about 2.4GB data during each iteration to broadcast the updated gradient. Without the optimization for low-rank matrix update, MatFast shuffles about 14.4GB data to store intermediate results. SystemML and MatFast generate a similar amount of data shuffle. MLlib performs the worst due to its inefficient implementation of matrix multiplications.

Sparse Matrix Chain Multiplications. We generate random matrices with the same dimensions to study the effectiveness of the dynamic cost-based optimization strategy. We generate matrix chains of length 4 ($\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \mathbf{A}_4$). Each matrix is of size 30,000 $\times$ 30,000. The skew type of input Matrix $\mathbf{A}_1$ and $\mathbf{A}_3$ are set for column-major fashion, and those of Matrix $\mathbf{A}_2$ and $\mathbf{A}_4$ are set for row-major fashion.

We fix the sparsity at $\rho = 0.01$ and vary the skew of the generated matrix data. Figure 9 gives the execution time and communication cost with median, minimum, and maximum w.r.t. various skew values. OPT is the optimal plan

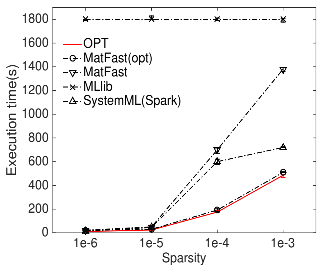


(a) Execution time

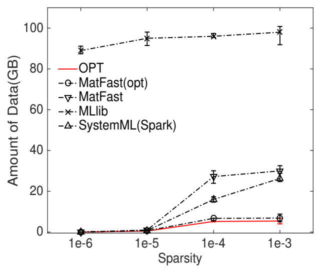


(b) Communication cost

Fig. 9: Costs of various skews for sparse matrix multiplication chain of length 4 with fixed sparsity $\rho = 0.01$



(a) Execution time

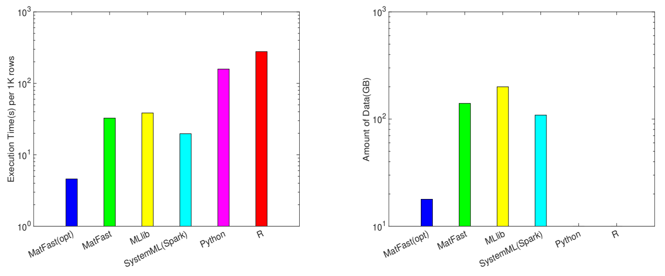


(b) Communication cost

Fig. 10: Costs of different sparsity values for matrix multiplication chains of length 4 with fixed skew $s = 0.5$

by enumerating all possible execution plans. MLlib evaluates the sparse matrix chain multiplication in a sequential order. It incurs high compute time and communication overheads due to its inability to perform sparse matrix multiplications in compressed format. MatFast with the optimizations turned off conducts sparse matrix chain multiplications in a sequential order. It outperforms MLlib because all the sparse matrix operations are executed over the compressed format. SystemML exploits worst-case estimation to predict the sparsity of the intermediate results. When the skew is small, i.e., $s = 5e$-4, the non-zero entries follow almost uniform distribution in the input matrices. The cost of the sequential execution plan is very close to the optimal. It takes MatFast(opt) extra overhead to collect the sampling statistics. Thus, MatFast(opt) performs worse than SystemML for extremely small skew values. However, dynamic cost estimation pays off as the skew becomes prominent. When the skew $s \geq 5e$-3, the optimal execution is no longer sequential. MatFast(opt) executes the multiplications in a nearly optimal order with negligible sampling costs.

Next, we fix the skew at $s = 0.5$ and vary the sparsity values. Figure 10 illustrates that when the sparsity value is very small, i.e., $\rho = 1e$-6, different plans have similar execution costs due to the extremely low sparsity in the matrices. MLlib performs the worst because it does not support sparse matrix multiplications in compressed format. As the sparsity increases, MatFast(opt) consistently outperforms SystemML with less execution time and communication cost. MatFast with the optimizations turned off performs slightly worse than SystemML when $\rho < 1e$-4. The sequential execution order adopted by MatFast incurs extensive computation costs when $\rho > 1e$-4 compared to SystemML.



(a) Execution time (b) Communication cost

Fig. 11: kruX algorithm for eQTL over multiple platforms

Biological Data Analysis. We evaluate matrix computations in a complex biological data analysis (Expression Quantitative Trait Loci (eQTL)) using the kruX technique [27]. The inputs are a genotype matrix (38,187,570 $\times$ 462) and an mRNA expression matrix (23,722 $\times$ 462). The output is a dense matrix (23,722 $\times$ 38,187,570) that takes about 7,250 GB of memory. This is far beyond our clusters' hardware. Thus, the whole genotype matrix is partitioned into chunks of size of $170,000 \times 462$, and the result takes about 32GB per chunk.

The kruX library provides serial implementations on popular platforms, e.g., Python and R. Figure 11 gives a performance comparison between the different platforms. We measure performance in terms of execution time per 1K rows of the result matrix. This metric is an important parameter for kruX to produce human checkable results. Both Python and R implementations run in a single node and do not leverage parallel compute resources. Python and R take 158s and 278s to produce 1K rows, respectively. In contrast, MatFast(opt) takes about 4.6s to compute 1K rows, while MatFast, MLlib and SystemML take 33s, 38.5s, and 20s, respectively, to compute 1K rows. The kruX algorithm involves various matrix operators, e.g., dense matrix multiplication with sparse matrix, matrix transpose, and element-wise operations. With an optimized execution plan and matrix data partitioning schemes, MatFast(opt) outperforms MLlib by 9 times and SystemML by 5 times for computing eQTL. MatFast(opt)'s communication cost is less than 10% of those for MLlib and SystemML.

Comparison with non-MapReduce-based Systems. To complete the performance study, we also compare MatFast with non-MapReduce-based systems, i.e., ScaLAPACK and SciDB. ScaLAPACK [13] is a library for high-performance linear algebra routines on distributed-memory message-passing computers. It provides interfaces for distributed matrix computation. SciDB [12] is a scalable database management system designed for advanced analytics on multi-dimensional array data model. SciDB provides SQL-like query interfaces for performing matrix operations, e.g., matrix multiplication and transpose.

Operation	ScaLAPACK	SciDB	MatFast(opt)
MG-dense	103s	706s	118s
MG-sparse	86s	566s	19s

TABLE VI: Comparison with ScaLAPACK and SciDB

We use matrix-matrix multiplications for comparing the performance of the various systems. We select a dense partition $\mathbf{G}$ from the genotype matrix with dimension 10,000 $\times$ 462, and the whole mRNA sampling Matrix $\mathbf{M}$ with dimension 23,722 $\times$ 462. To generate a sparse Matrix $\mathbf{G}_s$, we fix the sparsity to 0.01 and randomly select each entry in $\mathbf{G}$ with probability of 0.01 to fill out the corresponding position in $\mathbf{G}_s$. For SciDB, the sparse matrix multiplication is computed with the `spgemm` operator. ScaLAPACK, SciDB, and MatFast all run on the Hathi cluster, where each node launches eight processes.

Row "MG-dense" in Table VI gives the execution time for computing $\mathbf{M} \times \mathbf{G}^T$. ScaLAPACK and MatFast(opt)'s performance are comparable for dense matrix multiplications. However, ScaLAPACK does not provide any fault tolerance guarantees for big data computations. MatFast is built on top of Spark that naturally supports fault tolerance. SciDB is slower than the other two systems because it takes care of data placement on disk. SciDB redistributes the data on the compute workers to satisfy the requirement of ScaLAPACK. SciDB also incurs extra overhead for maintaining failure handling during execution. Row "MG-sparse" gives the execution time for computing $\mathbf{M} \times \mathbf{G}_s^T$. MatFast(opt) performs significantly better than ScaLAPACK and SciDB as it performs sparse matrix multiplication in compressed format. Both ScaLAPACK and SciDB are not well tuned for sparse matrix operations and they treat sparse matrices as dense ones.

VI. Related Work

Matrix computation has been an active research topic for many years in the high-performance computing (HPC) community. Existing libraries, e.g., BLAS [1] and LAPACK [3], provide efficient matrix operators. ScaLAPACK [7] is a distributed variant of LAPACK built on the SPMD (single program, multiple data) model, but it lacks support for sparse matrices, and is prone to machine failures. SpMachO [22] optimizes computation costs for sparse matrix chain multiplications in a single machine setting.

Many systems have been proposed to support efficient matrix computations using Hadoop [2] and Spark [34]. Pegasus [20], a Hadoop-based library, implements a special class of graph algorithms expressed in repeated matrix-vector multiplications. However, optimizing a single operation is restrictive for various matrix applications. Many systems provide interfaces for multiple matrix operations, e.g., HAMA [29], Mahout [4], MLI [30], MadLINQ [28], SystemML [18], and DMac [32]. HAMA and Mahout provide matrix algorithm implementations using the low-level MapReduce APIs that makes it hard to realize new algorithms and tune for performance. MLI is a programming interface built on top of Spark for ML applications. However, MLI does not provide optimizations for sequences of matrix computations.

MadLINQ [28] exploits fine-grained pipelining to explore inter-vertex parallelism. However, it lacks support for efficient sparse matrix operations. SystemML provides an R-like interface for matrix primitives, and translates the script into a series of optimized MapReduce jobs on Hadoop. However, SystemML lacks optimized data partitioning of input and intermediate matrices. A hybrid parallelization strategy [10] of combining task and data parallelism is proposed for SystemML to achieve comparable performance to in-memory computations. Column encoding schemes and operations over compressed matrices are proposed for SystemML [17] when data does not fit in memory. DMac [32] leverages matrix dependencies, but it does not identify the special features of a matrix program to reduce computation costs and memory footprints. SciDB [12] supports the array data model, but it treats each operator individually without tuning for a series of matrix operators. MATFAST differs from these platforms in that (1) MATFAST's optimized execution plan is generated progressively by leveraging dynamic sampling-based order selection for sparse matrix chain multiplications and rule-based heuristics for special features of a matrix program, (2) sparse matrix computations are conducted in the compressed formats, and (3) matrix data partitioning scheme assignment based on the optimized plan mitigates communication overhead in the distributed computing environment.

VII. CONCLUSIONS

MATFAST is an in-memory distributed platform that optimizes query pipelines of matrix operations. MATFAST takes advantage of both dynamic cost-based analysis and rule-based heuristics to generate an optimized query execution plan. The dynamic cost-based analysis leverages a sampling-based technique to estimate the sparsity of matrix chain multiplications. The rule-based heuristics explore the special features of a matrix program and these features are organized in a memory-efficient way. Furthermore, communication-efficient data partitioning schemes are applied to input and intermediate matrices based on a cost function for matrix programs. MATFAST has been implemented as a Spark library. The case studies and experiments on various matrix programs demonstrate that MATFAST achieves an order of magnitude performance gain compared to state-of-the-art systems.

ACKNOWLEDGMENT

This work was supported by an NPRP grant 4-1534-1-247 from the Qatar National Research Fund and by the National Science Foundation Grant IIS 1117766.

REFERENCES

[1] "Blas," http://www.netlib.org/blas/.

[2] "Hadoop," http://hadoop.apache.org/.

[3] "Lapack," http://www.netlib.org/lapack/.

[4] "Mahout," http://mahout.apache.org/.

[5] "Mllib," http://spark.apache.org/mllib/.

[6] "R," https://www.r-project.org/.

[7] "Scalapack," http://www.netlib.org/scalapack/.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[9] J. Bennett and S. Lanning, "The netflix prize," *KDD Cup*, pp. 35–35, 2007.

[10] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. R. Burdick, and S. Vaithyanathan, "Hybrid parallelization strategies for large-scale machine learning in systemml," *Proc. VLDB Endow.*, vol. 7, no. 7, pp. 553–564, Mar. 2014.

[11] M. Bohm, 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.*, vol. 37, no. 3, pp. 52–62, 2014.

[12] P. G. Brown, "Overview of scidb: Large scale array storage, processing and analysis," in *SIGMOD'10*. New York, NY, USA: ACM, 2010, pp. 963–968.

[13] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers in frontiers of massively parallel computation," in *Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[15] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04*. USENIX Association, 2004.

[17] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 960–971, Aug. 2016.

[18] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *ICDE'11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 231–242.

[19] M. I. Jordan, Ed., *Learning in Graphical Models*. Cambridge, MA, USA: MIT Press, 1999.

[20] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238.

[21] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2011.

[22] D. Kernert, F. Köhler, and W. Lehner, "Spmacho - optimizing sparse linear algebra expressions with probabilistic density estimation," in *EDBT*, 2015, pp. 289–300.

[23] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2006.

[24] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *NIPS'01*, T. Leen, T. Dietterich, and V. Tresp, Eds. MIT Press, 2001, pp. 556–562.

[25] L. Muchnik, S. Pei, L. C. Parra, S. D. S. Reis, J. S. Andrade Jr, S. Havlin, and H. A. Makse, "Origins of power-law degree distribution in the heterogeneity of human activity in social networks," *Sci. Rep.*, vol. 3, May 2013.

[26] J. Nocedal and S. J. Wright, *Numerical optimization*, ser. Springer Series in Operations Research and Financial Engineering. Berlin: Springer, 2006.

[27] J. Qi, H. Asl, J. Björkegren, and T. Michoel, "krux: matrix-based non-parametric eqtl discovery." *BMC Bioinformatics*, vol. 15, p. 11, 2014.

[28] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang, "Madlinq: Large-scale distributed matrix computation for the cloud," in *EuroSys'12*. New York, NY, USA: ACM, 2012, pp. 197–210.

[29] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *CLOUDCOM'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 721–726.

[30] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," in *ICDM*, 2013, pp. 1187–1192.

[31] S. K. Thompson, *Sampling*, ser. Wiley series in probability and statistics. New York: J. Wiley, 2002.

[32] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *SIGMOD'15*. New York, NY, USA: ACM, 2015, pp. 93–105.

[33] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. New York, NY, USA: Association for Computing Machinery and Morgan, 2016.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*. San Jose, CA: USENIX, 2012, pp. 15–28.

[35] S. Zhe, K. Zhang, P. Wang, K.-c. Lee, Z. Xu, Y. Qi, and Z. Ghahramani, "Distributed flexible nonlinear tensor factorization," in *Advances in Neural Information Processing Systems (NIPS)*, 2016.