DOI:10.1145/3318221

Compressed Linear Algebra for Declarative Large-Scale Machine Learning

By Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald

Abstract

Large-scale Machine Learning (ML) algorithms are often iterative, using repeated read-only data access and I/O-bound matrix-vector multiplications. Hence, it is crucial for performance to fit the data into single-node or distributed main memory to enable fast matrix-vector operations. General-purpose compression struggles to achieve both good compression ratios and fast decompression for block-wise uncompressed operations. Therefore, we introduce Compressed Linear Algebra (CLA) for lossless matrix compression. CLA encodes matrices with lightweight, value-based compression techniques and executes linear algebra operations directly on the compressed representations. We contribute effective column compression schemes, cache-conscious operations, and an efficient sampling-based compression algorithm. Our experiments show good compression ratios and operations performance close to the uncompressed case, which enables fitting larger datasets into available memory. We thereby obtain significant end-to-end performance improvements.

1. INTRODUCTION

Large-scale ML leverages large data collections to find interesting patterns or build robust predictive models.[7] Applications range from traditional regression, classification, and clustering to user recommendations and deep learning for unstructured data. The labeled data required to train these ML models is now abundant, thanks to feedback loops in data products and weak supervision techniques. Many ML systems exploit data-parallel frameworks such as Spark[20] or Flink[2] for parallel model training and scoring on commodity hardware. It remains challenging, however, to train ML models on massive labeled data sets in a cost-effective manner. We provide compression-based methods for accelerating the linear algebra operations that are central to training. The key ideas are to perform these operations directly on the compressed data, and to automatically determine the best lossless compression scheme, as required by declarative ML systems.

Declarative ML. State-of-the-art, large-scale ML systems provide high-level languages to express ML algorithms by means of linear algebra such as matrix multiplications, aggregations, element-wise and statistical operations. Examples at different abstraction levels are SystemML,[4] Mahout Samsara,[17] Spark MLlib,[19] and TensorFlow.[1] The high-level specification allows data scientists to create or customize ML algorithms without worrying about data and cluster characteristics, data representations (e.g., sparse or dense formats), and execution-plan generation.

Data-intensive ML algorithms. Many ML algorithms are iterative, with repeated read-only data access. These algorithms often rely on matrix-vector multiplications, which require one complete scan of the matrix with only two floating point operations per matrix element. This low operational intensity renders matrix-vector multiplication, even in-memory, I/O bound.[18] Despite the adoption of flash-and NVM-based SSDs, disk bandwidth is usually 10x–100x slower than memory bandwidth, which is in turn 10x–40x slower than peak floating point performance. Hence, it is crucial for performance to fit the matrix into available memory without sacrificing operations performance. This challenge applies to single-node in-memory computations, data-parallel frameworks with distributed caching like Spark,[20] and accelerators like GPUs with limited device memory. Even in the face of emerging memory and link technologies, the challenge persists due to increasing data sizes, different access costs in the memory hierarchy, and monetary costs.

Lossy versus lossless compression. Recently, lossy compression has received a lot of attention in ML. Many algorithms can tolerate a loss in accuracy because these algorithms are approximate in nature, and because compression introduces noise that can even improve the generalization of the model. Common techniques are (1) low- and ultra-low-precision storage and operations, (2) sparsification (which reduces the number of non-zero values), and (3) quantization (which reduces the value domain). However, these techniques require careful, manual application because they affect the accuracy in a data- and algorithm-specific manner. In contrast, declarative ML aims at physical data independence. Accordingly, we focus on lossless compression because it guarantees exact results and thus, it allows for *automatic* compression to fit large datasets in memory when needed.

Baseline solutions. The use of general-purpose compression techniques with block-wise decompression per operation is a common baseline solution. However, heavyweight techniques like Gzip are not applicable because decompression is too slow, while lightweight methods like Snappy or LZ4 achieve only modest compression ratios.

The original version of this paper was published in *PVLDB 9*, 12, 2016[9] and summarized in *SIGMOD Record 46*, 1, 2017[11]. This paper is based on an invited extended version that will appear in the *VLDB Journal*.[10]

Figure 1. Goals of compressed linear algebra.

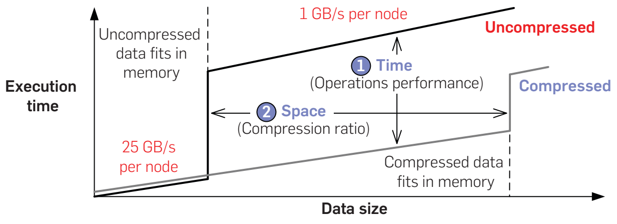
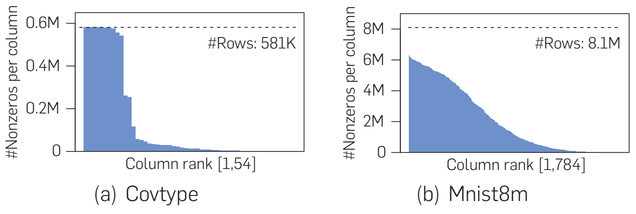

Execution time
Uncompressed data fits in memory
1 GB/s per node
Uncompressed
1 Time (Operations performance)
Compressed
2 Space (Compression ratio)
25 GB/s per node
Compressed data fits in memory
Data size

Figure 2. Sparsity skew.



Existing compressed matrix formats with good performance like CSR-VI[15] similarly show only moderate compression ratios. In contrast, our approach builds upon research on lightweight database compression, such as compressed bitmaps and dictionary coding, as well as sparse matrix representations.

Contributions. We introduce value-based Compressed Linear Algebra (CLA),[9,10] in which lightweight compression techniques are applied to matrices and then linear algebra operations are executed directly on the compressed representations. Figure 1 shows the goals of this approach: we want to widen the sweet spot for compression by achieving *both* (1) performance close to uncompressed in-memory operations, and (2) good compression ratios to fit larger datasets into memory. Our contributions include:

- Adapted column-oriented compression schemes for numeric matrices, and cache-conscious linear algebra operations over these compressed matrices (Section 3).
- A sampling-based algorithm for selecting a good compression plan, including techniques for compressed-size estimation and column grouping (Section 4).

Our CLA framework is available open source in Apache SystemML, where CLA is enabled by default for matrices that are larger than aggregate cluster memory.

2. BACKGROUND AND MOTIVATION

After giving an overview of SystemML as a representative ML system, we discuss common workload characteristics that directly motivate the design of our CLA framework.

SystemML compiler and runtime. In SystemML,[4] ML algorithms are expressed in a high-level language with R-like syntax for linear algebra and statistical operations. These scripts are automatically compiled into hybrid runtime plans that combine single-node, in-memory operations and distributed operations on MapReduce or Spark. During this compilation step, the system also applies optimizations such as common subexpression elimination, optimization of matrix-multiplication chains, algebraic simplifications, physical operator selection, and rewrites for dataflow properties like caching and partitioning. Matrices are represented in a binary *block matrix* format with fixed-size blocks, where individual blocks can be in dense, sparse, or ultra-sparse formats. For single-node operations, the entire matrix is represented as a block, which ensures consistency without unnecessary overheads. CLA can be seamlessly integrated by adding a new derived block representation and operations.

Common operation characteristics. Two important classes of ML algorithms are (1) iterative algorithms with matrix-vector multiplications (or matrix-matrix with a small second matrix), and (2) closed-form algorithms with transpose-self matrix multiplication. For both classes, few matrix operations dominate the overall algorithm runtime, apart from the costs for the initial read from distributed file system or object storage. This is especially true with hybrid runtime plans, where operations over small data are executed in the driver and thus, incur no latency for distributed computation. Examples for class (1) are linear regression via a conjugate gradient method (LinregCG), L2-regularized support vector machines (L2SVM), multinomial logistic regression (MLogreg), Generalized Linear Models (GLM), and Kmeans, while examples for class (2) are linear regression via a direct solve method (LinregDS) and Principal Component Analysis (PCA). Besides matrix-vector multiplication, we have vector-matrix multiplication, which is often caused by the rewrite $X^\top v \rightarrow (v^\top X)^\top$ to avoid transposing X because computing $X^\top$ is expensive, whereas computing $v^\top$ involves only a metadata update. Many systems also implement physical operators for matrix-vector chains with optional element-wise weighting $X^\top(w \odot (Xv))$, and transpose-self matrix multiplication (`tsmm`) $X^\top X$.[4,17] Most of these operations are I/O-bound, except for `tsmm` with $m \gg 1$ features because its compute workload grows as $O(m^2)$. Other common operations over X are `cbind`, unary aggregates like `colSums`, and matrix-scalar operations.

Common data characteristics. The inputs to these algorithm classes often exhibit common data characteristics:

- *Tall and skinny matrices:* Matrices usually have significantly more rows (observations) than columns (features), especially in enterprise ML, where data often originates from data warehouses (see Table 1).
- *Non-uniform sparsity:* Sparse datasets usually have many features, often created via pre-processing such as dummy coding. Sparsity, however, is rarely uniform, but varies among features. For example, Figure 2 shows the skew of the Covtype and Mnist8m datasets.
- *Low column cardinalities:* Many datasets exhibit features with few distinct values, for example, encoded categorical, binned or dummy-coded features. For example, Figure 3 shows the ratio of column cardinality to the number of rows of the Higgs and Census datasets.
- *Column correlations:* Correlation among features is also very common and typically originates from natural

Figure 3. Cardinality ratios.

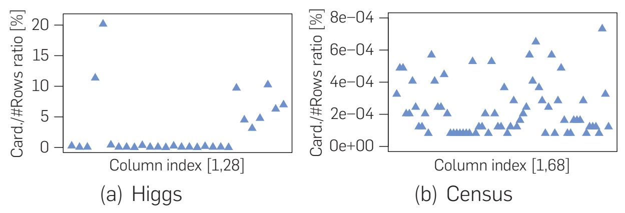
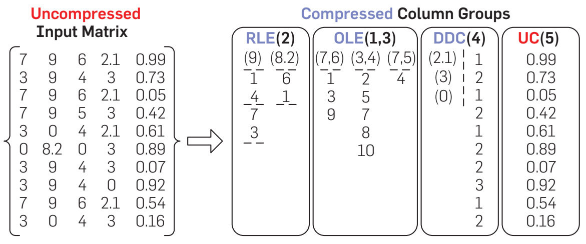


Figure 4. Example compressed matrix block.



data correlation, the use of composite features, or again pre-processing techniques like dummy coding. For example, exploiting column correlations improved the compression ratio for Census from 12.8x to 35.7x.

These data characteristics directly motivate the use of column-oriented compression schemes as well as heterogeneous encoding schemes and column co-coding.

3. COMPRESSION SCHEMES

We now describe the overall CLA compression framework, encoding formats for compressed column groups, and cache-conscious operations over compressed matrices.

3.1. Matrix compression framework

CLA compresses matrices column-wise to exploit two key characteristics: few distinct values per column and high cross-column correlations. Taking advantage of few distinct values, we encode a column as a *dictionary* of distinct values, and a list of *offsets* per value or value *references*. Offsets represent row indexes where a given value appears, while references encode values by their positions in the dictionary.

Column co-coding. We further exploit column correlation by partitioning columns into groups such that columns within each group are highly correlated. Each column group is then encoded as a single unit. Conceptually, each row of a column group comprising m columns is an m-tuple $\mathbf{t}$ of floating-point values that represent reals or integers.

Column encoding formats. The lists of offsets and references are then stored in a compressed representation. Inspired by database compression techniques and sparse matrix formats, we adapt four effective encoding formats:

- Offset-List Encoding (OLE) encodes the offset lists per value tuple as an ordered list of row indexes.
- Run-Length Encoding (RLE) encodes the offset lists as sequence of runs of begin row index and run length.
- Dense Dictionary Coding (DDC) stores tuple references to the dictionary including zeros.
- Uncompressed Columns (UC) is a fallback for incompressible columns, stored as a sparse or dense block.

Encoding may be heterogeneous, with different formats for different column groups. The decisions on co-coding and encoding formats are strongly data-dependent and thus, require automatic compression planning (Section 4).

Example compressed matrix. Figure 4 shows an example compressed matrix block in its logical representation. The 10 × 5 input matrix is encoded as four column groups, where we use 1-based indexes. Columns 2, 4, and 5 are represented as single-column groups and encoded via RLE, DDC, and UC, respectively. For Column 2 in RLE, we have two distinct non-zero values and hence two associated offset lists encoded as runs. Column 4 in DDC has three distinct values (including zero) and encodes the data as tuple references, whereas Column 5 is a UC group in dense format. Finally, there is a co-coded OLE column group for the correlated Columns 1 and 3, which encodes offset lists for all three distinct non-zero value-pairs as lists of row indexes.

Notation. For the ith column group, denote by $\mathcal{T}_i = \{\mathbf{t}_{i1}, \mathbf{t}_{i2}, \ldots, \mathbf{t}_{id_i}\}$ the set of d_i distinct tuples, by $\mathcal{G}_i$ the set of column indexes, and by $\mathcal{O}_{ij}$ the set of offsets associated with $\mathbf{t}_{ij}$ ($1 \leq j \leq d_i$). The OLE and RLE schemes are "sparse" formats in which zero values are not stored, whereas DDC is a dense format, which includes zero values. Also, denote by α the size in bytes of each floating point value, where $\alpha = 8$ for the double-precision IEEE-754 standard.

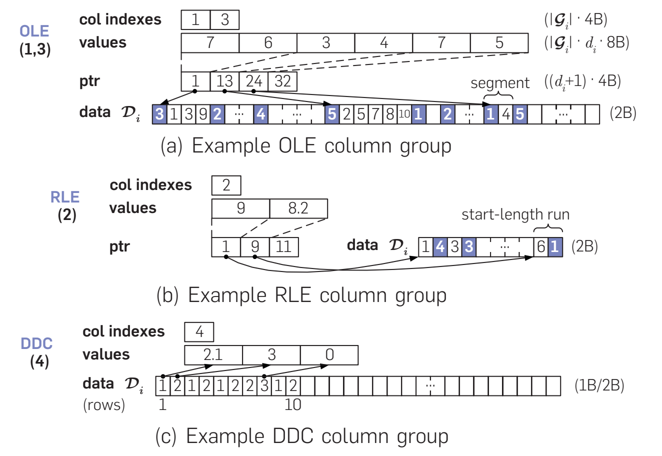
3.2. Column encoding formats

CLA uses heterogeneous encoding formats to exploit the full compression potential of individual columns. OLE and RLE use offset lists to map from value tuples to row indexes, while DDC uses tuple references to map from row indexes to value tuples. We now describe their physical data layouts.

Data layout. Figure 5 shows the data layouts of OLE, RLE, and DDC column groups for an extended example matrix (with more rows). All three formats use a common header of two arrays for column indexes and value tuples, as well as a data array $\mathcal{D}_i$. The header of OLE and RLE groups further contains an array for pointers to the data per tuple. The data length per tuple in $\mathcal{D}_i$ can be computed as the difference of adjacent pointers (e.g., for $\mathbf{t}_{i1} = (7, 6)$ as 13−1=12) because the offset lists are stored consecutively.

Offset-List Encoding (OLE). The OLE format divides the offset range into *segments* of fixed length $\Delta^s = 2^{16}$ to encode each offset with only two bytes. Offsets are mapped to their corresponding segments and encoded as the difference to the beginning of their segment. Each segment then stores the number of offsets followed by two bytes for each offset. For example, in Figure 5(a), the nine instances of (7, 6) appear in three consecutive segments

Figure 5. Data layout of encoding formats.



(a) Example OLE column group

(b) Example RLE column group

(c) Example DDC column group

with 3, 2, and 4 entries. Empty segments require two bytes indicating zero length. The size S_i^{OLE} of column group $\mathcal{G}_i$ is calculated as

$$S_i^{\text{OLE}} = 4|\mathcal{G}_i| + d_i\left(4 + \alpha|\mathcal{G}_i|\right) + 2\sum_{j=1}^{d_i} b_{ij} + 2z_i, \quad (1)$$

where b_{ij} is the number of segments of tuple $\mathbf{t}_{ij}$, $|\mathcal{O}_{ij}|$ is the number of offsets for $\mathbf{t}_{ij}$, and $z_i = \sum_{j=1}^{d_i} |\mathcal{O}_{ij}|$ is the total number of offsets—that is, the number of non-zero values—in the column group. The header size is $4|\mathcal{G}_i| + d_i(4 + \alpha|\mathcal{G}_i|)$.

Run-Length Encoding (RLE). RLE encodes ranges of offsets as a sequence of *runs*, where a run is stored as two bytes for both the starting offset and length. We use delta encoding to store the starting offset as its difference to the end of the preceding run. To ensure a two-byte representation, we store empty runs or partitioned runs when the starting offset or the run length exceed the maximum length of 2^{16}. The size S_i^{RLE} of column group $\mathcal{G}_i$ is calculated as

$$S_i^{\text{RLE}} = 4|\mathcal{G}_i| + d_i\left(4 + \alpha|\mathcal{G}_i|\right) + 4\sum_{j=1}^{d_i} r_{ij}, \quad (2)$$

where r_{ij} is the number of runs for tuple $\mathbf{t}_{ij}$.

Dense Dictionary Coding (DDC). The DDC format uses a dense, fixed-length data array $\mathcal{D}_i$ of n entries. The kth entry encodes the value tuple of the k^{th} row as its position in the dictionary. Therefore, the number of distinct tuples d_i in the dictionary determines the physical size per entry. We use two byte-aligned formats, DDC1 and DDC2, with one and two bytes per entry. Accordingly, these DDC formats are only applicable if $d_i \le 2^8$ or $d_i \le 2^{16}$. The total size S_i^{DDC} of column group $\mathcal{G}_i$ is then calculated as

$$S_i^{\text{DDC}} = \begin{cases} 4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i| + n & \text{if } d_i \le 2^8 \\ 4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i| + 2n & \text{if } 2^8 < d_i \le 2^{16}, \end{cases} \quad (3)$$

where $4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i|$ denotes the header size of column indexes and the dictionary of value tuples. In SystemML, we also share common dictionaries across DDC column groups, which is useful for image data in blocked matrix storage. Since OLE, RLE, and DDC are all value-based formats, column co-coding and common runtime techniques apply.

Limitations. An open research question is the handling of ultra-sparse matrices where our approach of empty OLE segments and RLE runs introduces substantial overhead.

3.3. Operations over compressed matrices

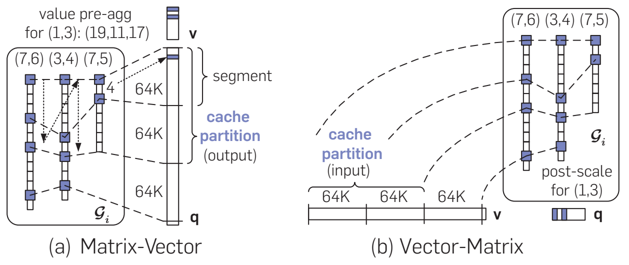
CLA executes linear algebra operations directly over a compressed matrix block, that is a set $\mathcal{X}$ of column groups. Composing these operations from group operations facilitates simplicity regarding heterogeneous formats. We write $c\mathbf{v}$, $\mathbf{u}{\cdot}\mathbf{v}$ and $\mathbf{u}\odot\mathbf{v}$ to denote element-wise scalar-vector multiplication, inner product, and element-wise vector product.

Exploiting the dictionary. Several operations can exploit the dictionary of distinct tuples to reduce the number of floating point operations. Examples are sparse-safe matrix-scalar operations such as $c\mathbf{X}$ that are computed only for distinct tuples, and unary aggregates such as colSums($\mathbf{X}$) that are computed based on counts per tuple. Matrix-vector and vector-matrix multiplications similarly exploit pre-aggregation and post-scaling. A straightforward way to implement matrix-vector multiply $\mathbf{q} = \mathbf{X}\mathbf{v}$ iterates over $\mathbf{t}_{ij}$ tuples per group, scanning $\mathcal{O}_{ij}$ and adding $\mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ at reconstructed offsets to $\mathbf{q}$, where $\mathbf{v}_{\mathcal{G}_i}$ is a subvector of $\mathbf{v}$ for the indexes in $\mathcal{G}_i$. However, the value-based representation allows pre-aggregating $\mathbf{u}_{ij} = \mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ once for each tuple $\mathbf{t}_{ij}$. The more columns co-coded and the fewer distinct tuples, the fewer floating point operations are required.

Matrix-vector multiplication. Despite pre-aggregation, pure column-wise processing would scan the $n \times 1$ output vector $\mathbf{q}$ once per value tuple, resulting in cache-unfriendly behavior for large n. We therefore use cache-conscious schemes for OLE and RLE groups based on *horizontal, segment-aligned scans*. As shown in Figure 6(a) for OLE, these horizontal scans allow bounding the working-set size of the output. Multi-threaded operations parallelize over segment-aligned partitions of rows $[rl, ru)$, which update independent ranges of $\mathbf{q}$. We find π_{ij}, the starting position of each $\mathbf{t}_{ij}$ in $\mathcal{D}_i$ by aggregating segment lengths until we reach rl. We further pre-compute $\mathbf{u}_{ij} = \mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ once for all tuples. For each cache partition of size Δ^c (such that $\Delta^c \cdot \alpha \cdot$ #cores fits in L3 cache, by default $\Delta^c = 2\Delta^s$), we then iterate over all distinct tuples but maintain the current positions π_{ij} as well. The inner loop then scans segments and adds $\mathbf{u}_{ij}$ via scattered writes at reconstructed offsets to the output $\mathbf{q}$. RLE is similarly realized except for sequential writes to $\mathbf{q}$ per run, special handling of partition boundaries, and additional state for reconstructed start offsets. In contrast, DDC does not require horizontal scans but allows—due to random access—cache blocking across multiple DDC groups, which we apply for DDC1 only because its temporary memory requirement for $\mathbf{u}_i$ is bounded by 2KB per group.

Example matrix-vector multiplication. As an example for OLE matrix-vector multiplication, consider the column group $\mathcal{G} = (1, 3)$ from Figure 4 and suppose that $\mathbf{v}_{\mathcal{G}} = (1, 2)$. For these two columns, uncompressed operations require 20 multiplications and 20 additions. Instead, we first pre-compute $\mathbf{u}_{ij}$ as $(7, 6) \cdot (1, 2) = 19$, $(3, 4) \cdot (1, 2) = 11$, and $(7, 5) \cdot (1, 2) = 17$.

Figure 6. Cache-conscious OLE operations.



Then, we iterate over segments per tuple and add these values at the reconstructed offsets to **q**. Specifically, we add 19 to **q**[i] for i = 1, 3, 9, then add 11 to **q**[i] for i = 2, 5, 7, 8, 10, and finally add 17 to **q**[i] for i = 4, 6. Due to co-coding and few distinct values, the compressed operation requires only 6 multiplications and 13 additions. Since addition is commutative and associative, the updates of individual column groups to **q** are independent.

Vector-matrix multiplication. Pure column-wise processing of vector-matrix would similarly suffer from cache-unfriendly behavior because we would scan the input vector **v** once for each distinct tuple. Our OLE/RLE group operations therefore again use *horizontal, segment-aligned scans* as shown in Figure 6(b). Here, we sequentially operate on cache partitions of **v**. The OLE, RLE, and DDC algorithms are similar to matrix-vector multiplication, but in the inner loop we sum up input-vector values according to the given offset list or references, and finally, scale the aggregates once with the values in $\mathbf{t}_{ij}$. For multi-threaded operations, we parallelize over column groups. The cache-partition size for OLE and RLE is equivalent to matrix-vector (by default $2\Delta^s$) except that RLE runs are allowed to cross partition boundaries due to group-wise parallelization.

Special matrix multiplications. We further support special matrix multiplications such as *matrix-vector multiplication chains* $\mathbf{p} = \mathbf{X}^\top(\mathbf{w}\odot(\mathbf{X}\mathbf{v}))$, and *transpose-self matrix multiplication* $\mathbf{R} = \mathbf{X}^\top\mathbf{X}$ by using the previously described column group operations on a per block level. For example, we effect $\mathbf{X}^\top\mathbf{X}$ by decompressing one column at a time and performing vector-matrix multiplications, exploiting the symmetry of the result to avoid redundant computation.

Limitations. Interesting research questions include efficient matrix-matrix multiplication and the automatic generation of fused operators over compressed matrices that match the performance of hand-coded CLA operations.

4. COMPRESSION PLANNING

Given an uncompressed $n \times m$ matrix block **X**, we automatically choose a compression plan, that is, a partitioning of compressible columns into column groups and a compression scheme per group. To keep the planning costs low, we provide sampling-based techniques for estimating the compressed size of an OLE, RLE, or DDC column group $\mathcal{G}_i$. Since exhaustive ($O(m^m)$) and brute-force greedy ($O(m^3)$) partitioning are infeasible, we further provide a bin-packing-based technique for column partitioning, and an efficient greedy algorithm with pruning and memoization for column grouping. Together, these techniques significantly reduce the number of candidate groups. Finally, we describe the compression algorithm including error corrections.

4.1. Estimating compressed size

For calculating the compressed size of a column group $\mathcal{G}_i$ with the formulas (1), (2), and (3), we need to estimate the number of distinct tuples d_i, non-zero tuples z_i, segments b_{ij}, and runs r_{ij}. Our estimators are based on a small sample of rows $\mathcal{S}$ drawn randomly and uniformly from **X** with $|\mathcal{S}| \ll n$. We have found that being conservative (overestimating compressed size) yields the most robust co-coding choices, so we make conservative choices in our estimator design.

Number of distinct tuples. Sampling-based estimation of the number of distinct tuples is a well studied but challenging problem. We use the *hybrid* estimator,[13] which is adequate compared to more expensive estimators. The idea is to estimate the degree of variability in the population frequencies of the tuples in $\mathcal{T}_i$ as low, medium, or high, based on the estimated squared coefficient of variation, and then apply a "generalized jackknife" estimator that performs well for the given variability regime. These estimators have the form $\hat{d} = d_{\mathcal{S}} + K(N^{(1)}/|\mathcal{S}|)$, where $d_{\mathcal{S}}$ is the number of distinct tuples in the sample, K is a constant computed from the sample, and $N^{(1)}$ is the number of "singletons," that is, the number of tuples that appear exactly once in $\mathcal{S}$.

Number of OLE segments. Not all elements of $\mathcal{T}_i$ will appear in the sample. Denote by $\mathcal{T}_i^o$ and $\mathcal{T}_i^u$ the sets of tuples observed and unobserved in the sample, and by d_i^o and d_i^u their cardinalities. The latter can be estimated as $\hat{d}_i^u = \hat{d}_i - d_i^o$. We also need to estimate the population frequencies of observed and unobserved tuples. Let f_{ij} be the population frequency of tuple $\mathbf{t}_{ij}$ and F_{ij} the sample frequency. A naïve estimate scales up F_{ij} to obtain $f_{ij}^{\text{naïve}} = (n/|\mathcal{S}|)F_{ij}$. Note that $\sum_{\mathbf{t}_{ij}\in\mathcal{T}_i^o} f_{ij}^{\text{naïve}} = n$ implies a zero population frequency for each unobserved tuple. We adopt a standard way of dealing with this issue and scale down the naïve frequency estimates by the estimated "coverage" C_i of the sample, defined as $C_i = \sum_{\mathbf{t}_{ij}\in\mathcal{T}_i^o} f_{ij}/n$. The usual estimator of coverage, originally due to Turing,[12] is

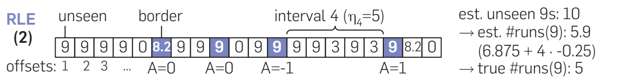
$$\hat{C}_i = \max\left(1 - N_i^{(1)}/|\mathcal{S}|, |\mathcal{S}|/n\right). \tag{4}$$

This estimator assumes a frequency of one for unseen tuples, computing the coverage as one minus the fraction of singletons $N_i^{(1)}$ in the sample. We add the lower sanity bound $|\mathcal{S}|/n$ to handle the special case $N_i^{(1)} = |\mathcal{S}|$. For simplicity, we assume equal frequencies for all unobserved tuples. The resulting frequency estimation formula for tuple $\mathbf{t}_{ij}$ is

$$\hat{f}_{ij} = \begin{cases} (n/|\mathcal{S}|)\hat{C}_i F_{ij} & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^o \\ n(1-\hat{C}_i)/\hat{d}_i^u & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^u. \end{cases} \tag{5}$$

We can now estimate the number of segments b_{ij} in which tuple $\mathbf{t}_{ij}$ appears at least once (this modified definition of b_{ij} ignores empty segments for simplicity with negligible error

Figure 7. Estimating the number of RLE runs $\hat{r}_{ij}$.



in our experiments). There are $l = n - |\mathcal{S}|$ unobserved offsets and estimated $\hat{f}^u_{iq} = \hat{f}_{iq} - F_{iq}$ unobserved instances of tuple $\mathbf{t}_{iq}$ for each $\mathbf{t}_{iq} \in \mathcal{T}_i$. We adopt a maximum-entropy (maxEnt) approach and assume that all assignments of unobserved tuple instances to unobserved offsets are equally likely. Denote by $\mathcal{B}$ the set of segment indexes and by $\mathcal{B}_{ij}$ the subset of indexes corresponding to segments with at least one observation of $\mathbf{t}_{ij}$. Also, for $k \in \mathcal{B}$, let l_k be the number of unobserved offsets in the kth segment and N_{ijk} the random number of unobserved instances of $\mathbf{t}_{ij}$ assigned to the k^{th} segment ($N_{ijk} \le l_k$). Set $Y_{ijk} = 1$ if $N_{ijk} > 0$ and $Y_{ijk} = 0$ otherwise. Then we estimate b_{ij} by its expected value $E[b_{ij}]$ under our maxEnt model:

$$\begin{aligned}\hat{b}_{ij} = E[b_{ij}] &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} P(N_{ijk} > 0) \\ &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} [1 - h(l_k, \hat{f}^u_{ij}, l)],\end{aligned} \tag{6}$$

where $h(a,b,c) = \binom{c-a}{a} \Big/ \binom{c}{a}$ is a hypergeometric probability. Note that $\hat{b}_{ij} \equiv \hat{b}^u_i$ for $\mathbf{t}_{ij} \in \mathcal{T}_i{}^u$, where $\hat{b}^u_i$ is the value of $\hat{b}_{ij}$ when $\hat{f}^u_{ij} = (1 - \hat{C}_i)\, n / \hat{d}^u_i$ and $|\mathcal{B}_{ij}| = 0$. Thus our estimate of the term $\sum_{j=1}^{d_i} b_{ij}$ in (1) is $\sum_{\mathbf{t}_{ij} \in \mathcal{T}_i^o} \hat{b}_{ij} + \hat{d}^u_i \hat{b}^u_i$.

Number of non-zero tuples. We estimate the number of non-zero tuples as $\hat{z}_i = n - \hat{f}_{i0}$, where $\hat{f}_{i0}$ is an estimate of the number of zero tuples in $\mathbf{X}_{:\mathcal{G}_i}$. Denote by F_{i0} the number of zero tuples in the sample. If $F_{i0} > 0$, we can proceed as above and set $\hat{f}_{i0} = (n/|\mathcal{S}|)\hat{C}_i F_{i0}$, where $\hat{C}_i$ is (4). If $F_{i0} = 0$, then we set $\hat{f}_{i0} = 0$; this estimate maximizes $\hat{z}_i$ and hence $\hat{S}^{\text{OLE}}_i$ per our conservative estimation strategy.

Number of RLE runs. The number of RLE runs r_{ij} for tuple $\mathbf{t}_{ij}$ is estimated as the expected value of r_{ij} under the maxEnt model. This expected value is very hard to compute exactly and Monte Carlo approaches are too expensive, so we approximate $E[r_{ij}]$ by considering one interval of consecutive unobserved offsets at a time as shown in Figure 7. Adjacent intervals are separated by a "border" comprising one or more observed offsets. As with the OLE estimates, we ignore the effects of empty and very long runs. Denote by η_k the length of the kth interval and set $\eta = \sum_k \eta_k$. Under the maxEnt model, the number f^u_{ijk} of unobserved $\mathbf{t}_{ij}$ instances assigned to the kth interval is hypergeometric, and we estimate f^u_{ijk} by its mean value: $\hat{f}^u_{ijk} = (\eta_k / \eta)\hat{f}^u_{ij}$. Given that $\hat{f}^u_{ijk}$ instances of $\mathbf{t}_{ij}$ are assigned randomly and uniformly among the η_k possible positions in the interval, the number of runs r_{ijk} within the interval (ignoring the borders) is known to follow a so-called "Ising-Stevens" distribution[14, pp. 422–423] and we estimate r_{ijk} by its mean: $\hat{r}_{ijk} = \hat{f}^u_{ijk}(\eta_k - \hat{f}^u_{ijk} + 1)/\eta_k$. A reasonable estimate for the contribution to r_{ij} from the border between intervals k and $k + 1$ is $\hat{A}_{ijk} = 1 - (2\hat{f}^u_{ij} / \eta)$.[10] Our final estimate for the number of runs is $\hat{r}_{ij} = \sum_k \hat{r}_{ijk} + \sum_k \hat{A}_{ijk}$.

Limitations. For ultra-sparse matrices, extended estimators are needed to account for empty segments and runs.

4.2. Partitioning columns into groups

To create column groups, we first divide compressible columns into independent partitions, and subsequently perform column grouping to find disjoint groups per partition. The overall objective is to maximize the compression ratio. Since exhaustive and brute-force grouping are infeasible, we focus on inexact but fast techniques.

Column partitioning. We observed empirically that column grouping usually generates small groups, and that the group extraction costs increase as the sample size, number of distinct tuples, or matrix density increases. These observations and the super-linear complexity of grouping motivate heuristics for column partitioning. Because data characteristics affect grouping costs, we use a *bin packing* strategy. The weight of the ith column is the cardinality ratio $\hat{d}_i / n$, indicating its effect on grouping costs. The capacity of a bin is a tuning parameter β, which ensures moderate grouping costs. Intuitively, bin packing creates a small number of bins with many columns per bin, which maximizes grouping potential while controlling processing costs. We made the design choice of a constant bin capacity—independent of z_i—to ensure constant compression throughput irrespective of blocking configurations. Finally, we solve this bin-packing problem with the first-fit decreasing heuristic.

Column grouping. A brute-force greedy method for column grouping starts with singleton groups and executes merging iterations. At each iteration, we merge the two groups yielding maximum compression ratio, that is, minimum change in size $\Delta\hat{S}_{ij} = \hat{S}_{ij} - \hat{S}_i - \hat{S}_j$. We terminate when no further size reductions are possible (i.e., no change in size $\Delta\hat{S}_{ij}$ is below 0). Although compression ratios are estimated from a sample, the cost of the naïve greedy method is $O(m^3)$. Our greedy algorithm additionally applies pruning and memoization. We execute merging iterations until the working set W reaches a fixpoint. In each iteration, we enumerate all $|W| \cdot (|W| - 1)/2$ candidate pairs of groups. A candidate can be safely pruned if any of its input groups has a size smaller than the currently best change in size $\Delta\hat{S}_{\text{opt}}$. This pruning threshold uses a natural lower bound $\underline{\hat{S}}_{ij} = \max(\hat{S}_i, \hat{S}_j)$ because at best the smaller group does not add any size. Substituting $\underline{\hat{S}}_{ij}$ into $\Delta\hat{S}_{ij}$ yields the lower bound $\Delta\underline{\hat{S}}_{ij} = -\min(\hat{S}_i, \hat{S}_j)$. Although this pruning does not change the worst-case complexity, it works very well in practice. Any remaining candidate is then evaluated, which entails extracting the column group from the sample and estimating its size $\hat{S}$. Observe that each merging iteration enumerates $O(|W|^2)$ candidates, but—ignoring pruning—only $O(|W|)$ candidates have not been evaluated in prior iterations; these are the ones formed by combining the previously merged group with each other element of $|W|$. Hence, we apply memoization to reuse statistics such as $\hat{S}_{ij}$, which reduces the complexity from $O(m^3)$ to $O(m^2)$ group extractions. Finally, we select a group and update the working set.

4.3. Compression algorithm

We now describe the matrix block compression algorithm (Algorithm 1). Note that we transpose the input in case of row-major dense or sparse formats to avoid performance issues due to repeated column-wise extraction.

Algorithm 1. Matrix Block Compression

```
Input:  Matrix block $\mathbf{X}$ of size $n \times m$
Output:  A set of compressed column groups $\mathcal{X}$
 1: $C^{C} \leftarrow \emptyset$, $C^{UC} \leftarrow \emptyset$, $\mathcal{G} \leftarrow \emptyset$, $\mathcal{X} \leftarrow \emptyset$
 2: // Planning phase – – – – – – – – – – – – – – – – – –
 3: $\mathcal{S} \leftarrow$ sampleRowsUniform($\mathbf{X}$, sample_size)
 4: parfor all columns $i$ in $\mathbf{X}$ do                    // classify
 5:     cmp_ratio $\leftarrow \hat{z}_i\alpha / \min\left(\hat{S}_i^{RLE}, \hat{S}_i^{OLE}, \hat{S}_i^{DDC}\right)$
 6:     if cmp_ratio $> 1$ then
 7:         $C^{C} \leftarrow C^{C} \cup i$
 8:     else
 9:         $C^{UC} \leftarrow C^{UC} \cup i$
10: bins $\leftarrow$ runBinPacking($C^{C}$)                       // group
11: parfor all bins $b$ in bins do
12:     $\mathcal{G} \leftarrow \mathcal{G} \cup$ greedyColumnGrouping($b$)
13: // Compression phase – – – – – – – – – – – – – – – – –
14: parfor all column groups $\mathcal{G}_i$ in $\mathcal{G}$ do            // compress
15:     do
16:         biglist $\leftarrow$ extractBigList($\mathbf{X}$, $\mathcal{G}_i$)
17:         cmp_ratio $\leftarrow$ getExactCmpRatio(biglist)
18:         if cmp_ratio $> 1$ then
19:             $\mathcal{X} \leftarrow \mathcal{X} \cup$ compressBigList(biglist), break
20:         $k \leftarrow$ removeLargestColumn($\mathcal{G}_i$)
21:         $C^{UC} \leftarrow C^{UC} \cup k$
22:     while $|\mathcal{G}_i| > 0$
23: return $\mathcal{X} \leftarrow \mathcal{X} \cup$ createUCGroup($C^{UC}$)
```

Planning phase (lines 2–12). Planning starts by drawing a sample of rows $\mathcal{S}$ from $\mathbf{X}$. For each column i, we first estimate the compressed column size S_i^C by $\hat{S}_i^C = \min(\hat{S}_i^{RLE}, \hat{S}_i^{OLE}, \hat{S}_i^{DDC})$, where $\hat{S}_i^{RLE}$, $\hat{S}_i^{OLE}$, and $\hat{S}_i^{DDC}$ are obtained by substituting the estimated $\hat{d}_i$, $\hat{z}_i$, $\hat{r}_{ij}$, and $\hat{b}_{ij}$ into formulas (1)–(3). We conservatively estimate the uncompressed column size as $\hat{S}_i^{UC} = \min(n\alpha, \hat{z}_i(4+\alpha))$, which covers both dense and sparse, with moderate underestimation for sparse as it ignores row pointers of sparse blocks, but this estimate allows column-wise decisions independent of $|C^{UC}|$. Columns whose estimated compression ratio $\hat{S}_i^{UC} / \hat{S}_i^C$ exceeds 1 are added to a compressible set C^C. In a last step, we divide the columns in C^C into bins and apply our greedy column grouping per bin to form column groups.

Compression phase (lines 13–23). The compression phase first obtains exact information about each column group and uses this information to adjust the groups, correcting for estimation errors. These exact statistics are also used to choose the optimal encoding format per column group. For each column group $\mathcal{G}_i$, we extract the "big" (i.e., uncompressed) list that comprises the set $\mathcal{T}_i$ of distinct tuples and uncompressed offsets per tuple. The big lists for all groups are extracted during a single column-wise pass through $\mathbf{X}$ using hashing. During this extraction operation, the parameters d_i, z_i, r_{ij}, and b_{ij} for each group $\mathcal{G}_i$ are computed exactly, with negligible overhead. These parameters are used in turn to calculate the exact compressed sizes $\hat{S}_i^{OLE}$, $\hat{S}_i^{RLE}$, and $\hat{S}_i^{DDC}$ with the formulas (1)–(3), and exact compression ratio $\hat{S}_i^{UC} / \hat{S}_i^C$ for each group.

Corrections. Because the column groups are originally formed using compression ratios that are estimated from a sample, there may be false positives, that is, purportedly compressible groups that are in fact incompressible. We attempt to correct such false-positive groups by iteratively removing the column with largest estimated size until the remaining group is either compressible or empty. Finally, the incompressible columns are collected into a single UC column group that is encoded in sparse or dense format.

Limitations. The temporary memory requirements of compression are negligible for distributed, block-wise processing but pose challenges for single-node environments.

5. EXPERIMENTS

We present selected, representative results from a broader experimental study.[9, 10] Overall, the experiments show that CLA achieves operations performance close to the uncompressed case while yielding substantially better compression ratios than lightweight general-purpose compression. Therefore, CLA provides large end-to-end performance improvements when uncompressed or lightweight-compressed matrices do not fit into aggregate cluster memory.

5.1. Experimental setting

Cluster setup. We ran all experiments on a 1+6 node cluster, that is, one head node of 2×4 Intel E5530 with 64 GB RAM, 6 worker nodes of 2×6 Intel E5-2440 with 96 GB RAM, 12×2 TB disks, and 10 GB Ethernet. We used Open-JDK 1.8.0, Apache Hadoop 2.7.3, and Apache Spark 2.1, in yarn-client mode, with 6 executors, 25 GB driver memory, 60 GB executor memory, and 24 cores per executor. Finally, we report results with Apache SystemML 0.14.

Implementation details. If CLA is enabled, SystemML automatically injects—for any multi-column input matrix—a so-called `compress` operator via rewrites, after initial read or text conversion but before checkpoints. The `compress` operator transforms an uncompressed into a compressed matrix block including compression planning. For distributed matrices, we compress individual blocks independently in a data-local manner. Making our compressed matrix block a subclass of the uncompressed matrix block yielded seamless compiler and runtime integration throughout SystemML.

5.2. Compression ratios and time

Compression ratios. Table 1 shows the compression ratios for the general-purpose, heavyweight Gzip, lightweight

Table 1. Compression ratios of real datasets.

	Size					
Dataset	**$n \times m$**	**sparsity**	**size**	**Gzip**	**Snappy**	**CLA**
Higgs[16]	11M × 28	0.92	2.5 GB	1.93	1.38	**2.17**
Census[16]	2.5M × 68	0.43	1.3 GB	17.11	6.04	**35.69**
Covtype[16]	581K × 54	0.22	0.14 GB	10.40	6.13	**18.19**
ImageNet[6]	1.3M × 900	0.31	4.4 GB	5.54	3.35	**7.34**
Mnist8m[5]	8.1M × 784	0.25	19 GB	4.12	2.60	**7.32**
Airline78[3]	14.5M × 29	0.73	3.3 GB	7.07	4.28	**7.44**

Snappy, and CLA on real datasets. Sizes are given as rows, columns, sparsity—that is, ratio of #non-zeros to cells—and in-memory size. We observe compression ratios of 2.2x–35.7x, due to a mix of floating point and integer data, and due to features with relatively few distinct values. Thus, ML datasets are indeed amenable to compression.

Compression and decompression. Overall, we observe reasonable average compression bandwidth across all datasets of roughly 100 MB/s (ranging from 67.7 MB/s to 184.4 MB/s), single-threaded. In comparison, the single-threaded compression throughput (including the time for matrix serialization) of the general-purpose Gzip and Snappy using native libraries, ranges from 6.9 MB/s to 35.6 MB/s and 156.8 MB/s to 353 MB/s, respectively. The decompression bandwidth (including the time for matrix deserialization) of Gzip ranges from 88 MB/s to 291 MB/s which is slower than for uncompressed I/O. Snappy achieves a decompression bandwidth between 232 MB/s and 638 MB/s. In contrast, CLA achieves good compression ratios and avoids decompression altogether.

5.3. Operations performance

Matrix-vector multiplication. Figure 8(a) shows the multi-threaded matrix-vector multiplication time. Despite row-wise updates of the output vector, CLA shows performance close to or better than ULA, except for Mnist8m and Airline78. The slowdown on the latter datasets is due to (1) many OLE tuple values, each requiring a pass over the output, and (2) the size of the output vector. For Mnist8m (8M rows) and Airline78 (14M rows), the output vectors do not fit into the L3 cache (15 MB). Accordingly, we see substantial improvements by cache-conscious CLA operations. ULA is a competitive baseline because it achieves peak single-socket/remote memory bandwidth of ≈25 GB/s. Multi-threaded CLA operations exhibit a speedup similar to ULA, in some cases even better: with increasing number of threads, ULA quickly saturates peak memory bandwidth, while CLA achieves improvements due to smaller bandwidth requirements and because multi-threading mitigates overheads. Figures 8(b) shows the vector-matrix multiplication time, where we see even better CLA performance because the column-wise updates favor CLA's column-wise layout.

Scalar and aggregate operations. As examples for exploiting the dictionary, Figures 8(c) and 8(d) show the results for the element-wise `X^2` and the unary aggregate `sum(X)`. Since `X^2` is executed over the dictionary only, we see speedups of three to five orders of magnitude, except for Higgs which has a large UC group with 9 out of 28 columns. Similarly, `sum(X)` is computed by efficient counting, which aggregates segment and run lengths, and subsequent scaling. We see improvements of up to 1.5 orders of magnitude compared to ULA, which is again at peak memory bandwidth.

5.4. End-to-End performance

RDD storage. ULA and CLA use the deserialized storage level `MEM_AND_DISK`, while Snappy and LZ4 use `MEM_AND_DISK_SER` because RDD compression requires serialized data. Table 2 shows the RDD storage size of Mnist8m with varying SystemML block size. For 16K, we observe compression ratios of 2.4x for Snappy and 2.5x for LZ4 but 5.6x for CLA. In contrast to the general-purpose schemes, CLA's compression advantage increases with larger block sizes because the common header is stored once per column group per block. SystemML 1.0 further shares DDC1 dictionaries across column groups if possible (CLA-SD), which makes CLA also applicable for small block sizes.

L2SVM on Mnist. SystemML compiles hybrid runtime plans, where only operations that exceed the driver memory are executed as Spark operations. For L2SVM, we have two scans of **X** per outer iteration (MV and VM), while all inner-loop operations are—equivalently for all baselines—executed in the driver. Figure 9 shows the results, where Spark evicts individual partitions of 128 MB, leading to a graceful performance degradation. As long as the data fits in memory (Mnist80m, 180 GB), all runtimes are almost identical, with Snappy/LZ4 and CLA showing overheads of up to 30% and 4%, respectively. However, if ULA no longer fits in memory

Figure 8. Selected operations performance.

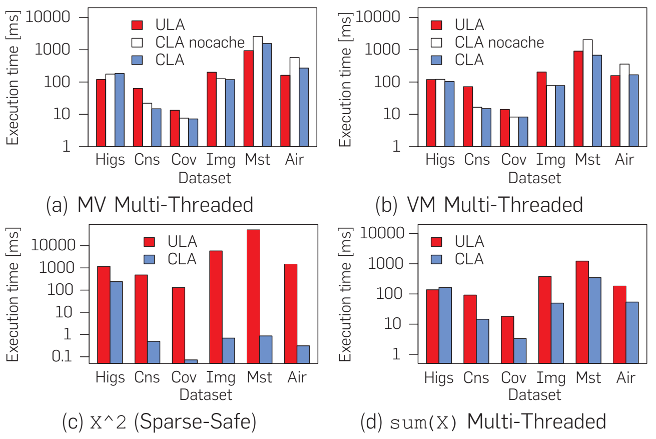


Table 2. Mnist8m RDD storage size.

Block Size	1,024	2,048	4,096	8,192	16,384
ULA	18 GB	18 GB	18 GB	18 GB	**18 GB**
Snappy	7.4 GB	7.4 GB	7.4 GB	7.4 GB	**7.4 GB**
LZ4	7.1 GB	7.1 GB	7.1 GB	7.1 GB	**7.1 GB**
CLA	7.9 GB	5.6 GB	4.8 GB	3.8 GB	**3.2 GB**
CLA-SD	4.3 GB	3.6 GB	3.5 GB	3.3 GB	**3 GB**

Figure 9. L2SVM end-to-end performance Mnist.

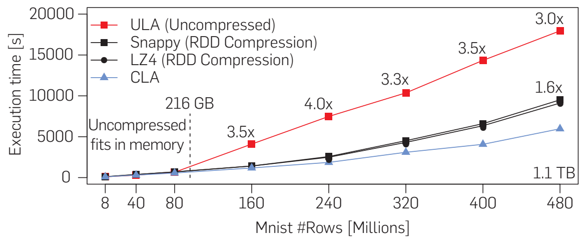


Table 3. ML algorithms (https://systemml.apache.org/algorithms) end-to-end performance Mnist40m/240m/480m.

	Mnist40m (90 GB)			Mnist240m (540 GB)			Mnist480m (1.1 TB)		
Algorithm	**ULA**	**Snappy**	**CLA**	**ULA**	**Snappy**	**CLA**	**ULA**	**Snappy**	**CLA**
L2SVM	**296s**	386s	308s	7,483s	2,575s	**1,861s**	17,950s	9,510s	**5,973s**
Mlogreg	490s	665s	**463s**	18,146s	5,975s	**3,240s**	71,140s	26,998s	**12,653s**
GLM	346s	546s	**340s**	17,244s	4,148s	**2,183s**	61,425s	20,317s	**10,418s**
LinregCG	**87s**	135s	93s	3,496s	765s	**463s**	6,511s	2,598s	**986s**
LinregDS	**79s**	148s	145s	1,080s	798s	**763s**	2,586s	1,954s	**1,712s**
PCA	**76s**	140s	146s	**711s**	760s	730s	1,551s	1,464s	**1,412s**

(Mnist160m, 360 GB), compression leads to significant improvements because the good compression ratio of CLA allows fitting larger datasets into memory.

Other ML algorithms on Mnist. Table 3 further shows results for a range of algorithms—including algorithms with RDD operations in nested loops (e.g., GLM, Mlogreg) and non-iterative algorithms (e.g., LinregDS and PCA)—for the interesting points of Mnist40m (90 GB), where all datasets fit in memory, Mnist240m (540 GB), and Mnist480m (1.1 TB). For Mnist40m and iterative algorithms, we see similar ULA/CLA performance but a slowdown of up to 57% with Snappy. This is because RDD compression incurs decompression overhead per iteration. For non-iterative algorithms, CLA and Snappy show overheads of up to 92% and 87%, respectively. Beside the initial compression overhead, CLA also shows less efficient `tsmm` performance. For iterative algorithms over Mnist240m and Mnist480, we see significant performance improvements by CLA. This is due to many inner iterations with RDD operations in the outer and inner loops and thus, less read.

Code generation. With CLA, the bottleneck partially shifted to the driver operations. Code generation for operator fusion[8] further improves the L2SVM runtime to 181 s/1,068 s/3,565 s, increasing the relative benefits of CLA.

6. CONCLUSION

To summarize, CLA compresses matrices with lightweight value-based compression techniques—inspired by database compression and sparse matrix formats—and performs linear algebra operations directly over the compressed representation. We introduced effective column encoding schemes, cache-conscious operations, and an efficient sampling-based compression algorithm. Our experiments show good compression ratios and fast operations close to the uncompressed case, which provides significant performance benefits when data does not fit into memory. Therefore, CLA is used by default for large matrices in SystemML, but it is also broadly applicable to any system that provides blocked matrix representations, linear algebra, and physical data independence.

References

1. Abadi, M. et al. TensorFlow: A system for large-scale machine learning. In *OSDI* (2016).
2. Alexandrov, A. et al. The stratosphere platform for big data analytics. *VLDB J. 23*, 6 (2014).
3. American Statistical Association (ASA). Airline on-time performance dataset. stat-computing.org/dataexpo/2009.
4. Boehm, M., et al. SystemML: Declarative machine learning on spark. *PVLDB 9*, 13 (2016).
5. Bottou, L. The infinite MNIST dataset. leon.bottou.org.
6. Chitta, R. et al. Approximate Kernel k-means: Solution to large scale Kernel clustering. In *KDD* (2011).
7. Cohen, J. et al. MAD skills: New analysis practices for big data. *PVLDB 2*, 2 (2009).
8. Elgamal, T. et al. SPOOF: Sum-product optimization and operator fusion for large-scale machine learning. In *CIDR* (2017).
9. Elgohary, A. et al. Compressed linear algebra for large-scale machine learning. *PVLDB 9*, 12 (2016).
10. Elgohary, A. et al. Compressed linear algebra for large-scale machine learning. *VLDB J.* (2017a). https://doi.org/10.1007/s00778-017-0478-1.
11. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., and Reinwald, B. Scaling Machine Learning via Compressed Linear Algebra. *SIGMOD Record 46*, 1 (2017b).
12. Good, I.J. The population frequencies of species and the estimation of population parameters. Biometrika (1953).
13. Haas, P.J. and Stokes, L. Estimating the number of classes in a finite population. JASA *93*, 444 (1998).
14. Johnson, N.L. et al. *Univariate Discrete Distributions*, 2nd edn. Wiley, New York, 1992.
15. Kourtis, K. et al. Optimizing sparse matrix-vector multiplication using index and value compression. In CF (2008).
16. Lichman, M. UCI machine learning repository: Higgs, covertype, US census (1990). archive.ics.uci.edu/ml/.
17. Schelter, S. et al. Samsara: Declarative machine learning on distributed dataflow systems. NIPS MLSystems (2016).
18. Williams, S. et al. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM 52*, 4 (2009).
19. Zadeh, R.B. et al. Matrix computations and optimization in apache spark. In *KDD* (2016).
20. Zaharia, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012).

Ahmed Elgohary, University of Maryland, College Park, MD, USA.

Matthias Boehm, Frederick R. Reiss, and Berthold Reinwald, IBM Research—Almaden, San Jose, CA, USA.

Peter J. Haas, University of Massachusetts, Amherst, MA, USA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/compressed-linear-algebra