Compressed Linear Algebra for Large-Scale Machine Learning

Ahmed Elgohary[2]*, Matthias Boehm[1], Peter J. Haas[1], Frederick R. Reiss[1], Berthold Reinwald[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] University of Maryland; College Park, MD, USA

ABSTRACT

Large-scale machine learning (ML) algorithms are often iterative, using repeated read-only data access and I/O-bound matrix-vector multiplications to converge to an optimal model. It is crucial for performance to fit the data into single-node or distributed main memory. General-purpose, heavy- and lightweight compression techniques struggle to achieve both good compression ratios and fast decompression speed to enable block-wise uncompressed operations. Hence, we initiate work on compressed linear algebra (CLA), in which lightweight database compression techniques are applied to matrices and then linear algebra operations such as matrix-vector multiplication are executed directly on the compressed representations. We contribute effective column compression schemes, cache-conscious operations, and an efficient sampling-based compression algorithm. Our experiments show that CLA achieves in-memory operations performance close to the uncompressed case and good compression ratios that allow us to fit larger datasets into available memory. We thereby obtain significant end-to-end performance improvements up to 26x or reduced memory requirements.

1. INTRODUCTION

Data has become a ubiquitous resource [16]. Large-scale machine learning (ML) leverages these large data collections in order to find interesting patterns and build robust predictive models [16, 19]. Applications range from traditional regression analysis and customer classification to recommendations. In this context, often data-parallel frameworks such as MapReduce [20], Spark [51], or Flink [2] are used for cost-effective parallelization on commodity hardware.

Declarative ML: State-of-the-art, large-scale ML aims at declarative ML algorithms [12], expressed in high-level languages, which are often based on linear algebra, i.e., matrix multiplications, aggregations, element-wise and statistical operations. Examples—at different abstraction levels—are SystemML [21], SciDB [44], Cumulon [27], DMac [50], and TensorFlow [1]. The high level of abstraction gives data scientists the flexibility to create and customize ML algorithms independent of data and cluster characteristics, without worrying about the underlying data representations (e.g., sparse/dense format) and execution plan generation.

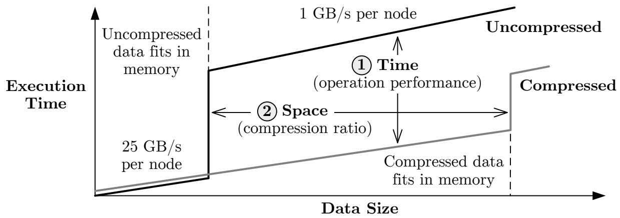


Figure 1: Goals of Compressed Linear Algebra.

Problem of Memory-Centric Performance: Many ML algorithms are iterative, with repeated read-only access to the data. These algorithms often rely on matrix-vector multiplications to converge to an optimal model. Matrix-vector multiplications are I/O-bound because they require one complete scan of the matrix, but only two floating point operations per matrix element. Hence, it is crucial for performance to fit the matrix into available memory because memory bandwidth is usually 10x-100x higher than disk bandwidth (but, for matrix-vector, still 10x-40x smaller than peak floating point performance, and thus, matrix-vector remains I/O-bound). This challenge applies to single-node in-memory computations [28], data-parallel frameworks with distributed caching such as Spark [51], and hardware accelerators like GPUs, with limited device memory [1, 4, 7].

Goals of Compressed Linear Algebra: Declarative ML provides data independence, which allows for automatic compression to fit larger datasets into memory. A baseline solution would be to employ general-purpose compression techniques and decompress matrices block-wise for each operation. However, heavyweight techniques like Gzip are not applicable because decompression is too slow, while lightweight methods like Snappy only achieve moderate compression ratios. Existing special-purpose compressed matrix formats with good performance like CSR-VI [34] similarly show only modest compression ratios. Our approach builds upon research on lightweight database compression, such as compressed bitmaps, and sparse matrix representations. Specifically, we initiate the study of *compressed linear algebra (CLA)*, in which database 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.

*Work done during an internship at IBM Research – Almaden.

Proceedings of the VLDB Endowment, Vol. 9, No. 12
Copyright 2016 VLDB Endowment 2150-8097/16/08.

Table 1: Compression Ratios of Real Datasets.

Dataset	Size	Gzip	Snappy	CLA
Higgs [37]	11M×28, 0.92: 2.5 GB	1.93	1.38	**2.03**
Census [37]	2.5M×68, 0.43: 1.3 GB	17.11	6.04	**27.46**
Covtype [37]	600K×54, 0.22: .14 GB	10.40	6.13	**12.73**
ImageNet [15]	1.2M×900, 0.31: 4.4 GB	5.54	3.35	**7.38**
Mnist8m [13]	8.1M×784, 0.25: 19 GB	4.12	2.60	**6.14**

Compression Potential: Our focus is on floating-point matrices (with 53/11 bits mantissa/exponent), so the potential for compression may not be obvious. Table 1 shows compression ratios for the general-purpose, heavyweight Gzip and lightweight Snappy algorithms and for our CLA method on real-world datasets (sizes given as rows, columns, sparsity, and in-memory size). We see compression ratios of 2x-27x, due to the presence of a mix of floating point and integer data, and due to features with relatively few distinct values. Thus, unlike in scientific computing [10], enterprise machine-learning datasets are indeed amenable to compression. The decompression bandwidth (including 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 but only moderate compression ratios. In contrast, CLA achieves good compression ratios and avoids decompression.

Contributions: Our major contribution is to make a case for *compressed linear algebra*, where linear algebra operations are directly executed over compressed matrices. We leverage ideas from database compression techniques and sparse matrix representations. The novelty of our approach is a combination of both, leading towards a generalization of sparse matrix representations and operations. The structure of the paper reflects our detailed technical contributions:

- *Workload Characterization:* We provide the background and motivation for CLA in Section 2 by giving an overview of Apache SystemML, and describing typical linear algebra operations and data characteristics.
- *Compression Schemes:* We adapt several column-based compression schemes to numeric matrices in Section 3 and describe efficient, cache-conscious core linear algebra operations over compressed matrices.
- *Compression Planning:* In Section 4, we further provide an efficient sampling-based algorithm for selecting a good compression plan, including techniques for compressed-size estimation and column grouping.
- *Experiments:* Finally, we integrated CLA into Apache SystemML. In Section 5, we study a variety of full-fledged ML algorithms and real-world datasets in both single-node and distributed settings. We also compare CLA against alternative compression schemes.

2. BACKGROUND AND MOTIVATION

In this section, we provide the background and motivation for compressed linear algebra. After giving an overview of SystemML as a representative platform for declarative ML, we discuss common workload and data characteristics. We also provide further evidence of compression potential.

2.1 SystemML Architecture

SystemML [21] aims at declarative ML [12], where algorithms are expressed in a high level scripting language having an R-like syntax and compiled to hybrid runtime plans that combine both single-node, in-memory operations and distributed operations on MapReduce or Spark [28]. We outline the features of SystemML relevant to CLA.

ML Program Compilation: An ML script is first parsed into a hierarchy of statement blocks and statements, where blocks are delineated by control structures such as loops and branches. Each statement block is translated to a DAG of high-level operators, and the system then applies various rewrites, such as common subexpression elimination, optimization of matrix-multiplication chains, algebraic simplification, and rewrites for dataflow properties such as caching and partitioning. Information about data size and sparsity are propagated from the inputs through the entire program to enable worst-case memory estimates per operation. These estimates are used during an operator-selection step, yielding a DAG of low-level operators, which is then compiled into a runtime program of executable instructions.

Distributed Matrix Representations: SystemML supports various input formats, all of which are internally converted into a binary *block matrix* format with fixed-size blocks. Similar structures, called tiles [27], chunks [44], or blocks [21], are widely used in existing large-scale ML systems. Each block is represented either in dense or sparse format in order to allow for block-local decisions and efficiency on datasets with non-uniform sparsity. SystemML uses a modified CSR (compressed sparse row) format for sparse matrix blocks. For single-node, in-memory operations, the entire matrix is often represented as a single block [28], which allows reuse of block operations across runtime backends. CLA can be seamlessly integrated by adding a new derived block representation and operations. We provide further details of CLA in SystemML in Section 5.1.

2.2 Workload Characteristics

We now describe common ML workload characteristics in terms of linear algebra operations and properties of matrices.

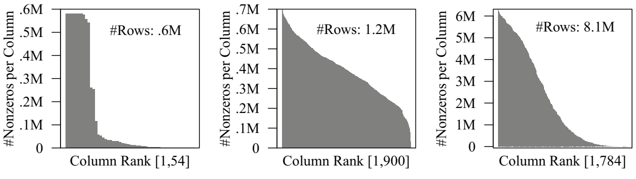
An Example: Consider the task of fitting a simple linear regression model via the conjugate gradient (CG) method [4, 21, 50]. The LinregCG algorithm reads matrix **X** and vector **y**, including metadata from HDFS, and iterates CG steps until the error—as measured by an appropriate norm—falls below a target value. The ML script looks as follows:

```
X = read($1);                    # n x m feature matrix
y = read($2);                    # n x 1 label vector
maxi = 50; lambda = 0.001;  ...
r = -(t(X) %*% y);               # %*%..matrix multiply
norm_r2 = sum(r * r); p = -r;   # initial gradient
w = matrix(0, ncol(X), 1); i = 0;
while(i < maxi & norm_r2 > norm_r2_trgt) {
   # compute conjugate gradient
   q = ((t(X) %*% (X %*% p)) + lambda * p);
   # compute step size
   alpha = norm_r2 / sum(p * q);
   # update model and residuals
   w = w + alpha * p;
   r = r + alpha * q;
   old_norm_r2 = norm_r2;
   norm_r2 = sum(r^2); i = i + 1;
   p = -r + norm_r2/old_norm_r2 * p; }
write(w, $3, format="text");
```

Common Operation Characteristics: Two important classes of ML algorithms are (1) iterative algorithms with matrix-vector multiplications as above, and (2) closed-form algorithms with transpose-self matrix multiplication. For both classes, a small number of matrix operations dominate the overall algorithm runtime (apart from initial read

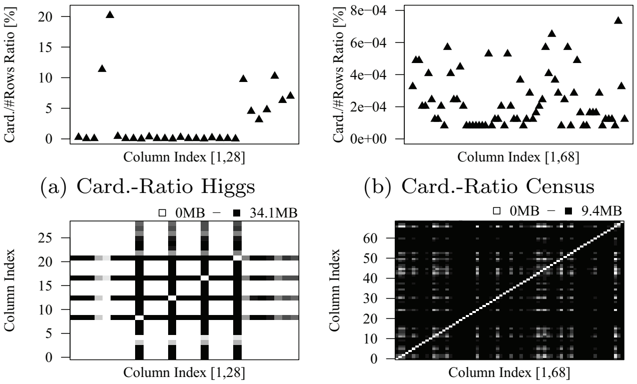
Table 2: Overview ML Algorithm Core Operations (see `http://systemml.apache.org/algorithms` for details).

Algorithm	M-V $\mathbf{Xv}$	V-M $\mathbf{v}^\top\mathbf{X}$	MVChain $\mathbf{X}^\top(\mathbf{w}\odot(\mathbf{Xv}))$	TSMM $\mathbf{X}^\top\mathbf{X}$
LinregCG	✓	✓	✓ (w/o $\mathbf{w}\odot$)	
LinregDS		✓		✓
Logreg / GLM	✓	✓	✓ (w/ $\mathbf{w}\odot$)	
L2SVM	✓	✓		
PCA	✓			✓



(a) Covtype (b) ImageNet (c) Mnist8m

Figure 2: Sparsity Skew (Non-Uniform Sparsity).



(a) Card.-Ratio Higgs (b) Card.-Ratio Census

(c) Co-Coding Higgs (d) Co-Coding Census

Figure 3: Cardinality Ratios and Co-Coding.

costs). This is especially true with hybrid runtime plans (see Section 2.1), where single-node operations over small data incur no latency for distributed computation. In LinregCG, for example, only lines 4 and 9 access matrix $\mathbf{X}$; all other computations are inexpensive operations over small vectors or scalars. Table 2 summarizes the core operations of important ML algorithms. Besides matrix-vector multiplication (e.g., line 9), we have vector-matrix multiplication, often caused by the rewrite $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$ to avoid transposing $\mathbf{X}$ (e.g., lines 4 and 9) because $\mathbf{X}^\top$ is very expensive, while dense vector transpositions are realized as pure metadata operations per block. In addition, many systems also implement physical operators for matrix-vector chains, with optional element-wise weighting $\mathbf{w}\odot$ (e.g., line 9), and transpose-self matrix multiplication (TSMM) $\mathbf{X}^\top\mathbf{X}$ [4, 28]. All of these operations are I/O-bound, except for TSMM with $m \gg 1$ features because its compute workload grows as $O(m^2)$. Beside these operations, `append`, unary aggregates like `colSums`, and matrix-scalar operations access $\mathbf{X}$ for intercept computation, scaling and shifting.

Common Data Characteristics: Despite significant differences in data sizes—ranging from kilobytes to terabytes—we and others have observed common data characteristics for the aforementioned algorithm classes:

- *Tall and Skinny Matrices:* Matrices usually have significantly more rows (observations) than columns (features), especially in enterprise machine learning [4, 52], where data often originates from data warehouses.
- *Non-Uniform Sparsity:* Sparse datasets usually have many features, often created via feature pre-processing like dummy coding. Sparsity, however, is rarely uniform, but varies among features. For example, Figure 2 shows the sparsity skew of three sparse datasets.
- *Low Column Cardinalities:* Many datasets exhibit features with few distinct values, e.g., encoded categorical, binned or dummy-coded (0/1) features.

The foregoing three data characteristics directly motivate the use of column-based compression schemes.

2.3 Compression Potential and Strategy

Examination of the datasets from Table 1 shows that *column cardinality* and *column correlation* should be key drivers of a column-based compression strategy.

Column Cardinality: The ratio of column cardinality (number of distinct values) to the number of rows is a good indicator of compression potential because it quantifies redundancy, independently of the actual values. Figures 3(a) and 3(b) show the ratio of column cardinality to the number of rows (in %) per column in the datasets *Higgs* and *Census*. All columns of *Census* have a cardinality ratio below .0008% and the majority of columns of *Higgs* have a cardinality ratio below 1%. There is also skew in the column cardinalities; for example, *Higgs* contains several columns having millions of distinct values. These observations motivate value-centric compression with fallbacks for high cardinality columns.

Column Correlation: Another indicator of compression potential is the correlation between columns with respect to the number of distinct value-pairs. With value-based offset lists, a column i with d_i distinct values requires $\approx 8d_i + 4n\,\mathrm{B}$, where n is the number of rows, and each value is encoded with 8 B and a list of 4 B row indexes. Co-coding two columns i and j as a single group of value-pairs and offsets requires $16d_{ij} + 4n\,\mathrm{B}$, where d_{ij} is the number of distinct value-pairs. The larger the correlation, the larger the size reduction by co-coding. Figures 3(c) and 3(d) show the size reductions (in MB) by co-coding all pairs of columns of *Higgs* and *Census*. For *Higgs*, co-coding any of the columns 8, 12, 16, and 20 with one of *most* of the other columns reduces sizes by at least 25 MB. Moreover, co-coding *any* column pair of *Census* reduces sizes by at least 9.3 MB. Overall, co-coding column groups of *Census* (not limited to pairs) improves the compression ratio from 10.1x to 27.4x. We therefore endeavor to discover and co-code column groups.

3. COMPRESSION SCHEMES

We now describe our novel matrix compression framework, including two effective encoding formats for compressed column groups, as well as efficient, cache-conscious core linear algebra operations over compressed matrices.

3.1 Matrix Compression Framework

As motivated in Sections 2.2 and 2.3, we represent a compressed matrix block as a set of compressed columns. Column-wise compression leverages 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 list of distinct values together with a list of *offsets* per value, i.e., a list of row indexes in which the value appears. We shall show that, similar to sparse matrix formats, offset lists allow for efficient linear algebra operations.

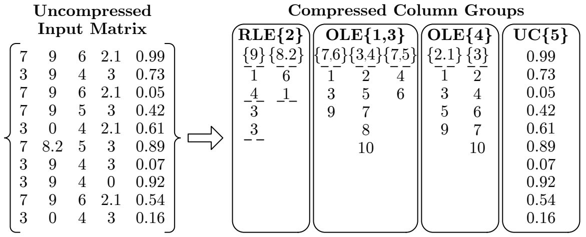


Figure 4: Example Compressed Matrix Block.

Column Co-Coding: We exploit column correlation—as discussed in Section 2.3—by partitioning columns into *column groups* such that columns within each group are highly correlated. Columns within the same group are then co-coded as a single unit. Conceptually, each row of a column group comprising m columns is an m-tuple $\mathbf{t}$ of floating-point values, representing reals or integers.

Column Encoding Formats: Conceptually, the offset list associated with each distinct tuple is stored as a compressed sequence of bytes. The efficiency of executing linear algebra operations over compressed matrices strongly depends on how fast we can iterate over this compressed representation. We adapt two well-known effective offset-list encoding formats: Offset-List Encoding (OLE) and Run-Length Encoding (RLE), as well as uncompressed columns (UC) as a fallback if compression is not beneficial. These decisions on column encoding formats as well as co-coding are strongly data-dependent and hence require automatic optimization. We discuss compression planning in Section 4.

Example Compressed Matrix: Figure 4 shows our running example of a compressed matrix block. The 10 × 5 input matrix is represented as four column groups, where we use 1-based indexing. Columns 2, 4, and 5 are represented as single-column groups and encoded with RLE, OLE, and UC, respectively. For column 4, we have two distinct non-zero values and hence two offset lists. Finally, there is a co-coded column group for the correlated columns 1 and 3, which encodes offset lists for all distinct value-pairs.

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 non-zero tuples, by $\mathcal{G}_i$ the set of column indexes, and by $\mathcal{O}_{ij}$ the set of offsets associated with $\mathbf{t}_{ij}$ $(1 \leq j \leq d_i)$. We focus on the "sparse" case in which zero values are not stored (0-suppressing). 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

Our framework relies on the complementary OLE, RLE, and UC representations. We now describe the compressed data layout of these formats and give formulas for the in-memory compressed size S_i^{OLE} and S_i^{RLE}. The total matrix size is then computed as the sum of group size estimates.

Data Layout: Figure 5 shows—as an extension to our running example from Figure 4 (with more rows)—the data layout of OLE/RLE column groups composed of four linearized arrays. Both encoding schemes use a common header of three arrays for column indexes, fixed-length value tuples, and pointers to the data per tuple as well as a data array $\mathcal{D}_i$. The physical 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). The data array is then used in an encoding-specific manner. Tuples are stored in order of decreasing physical data length to improve branch prediction and pre-fetching.

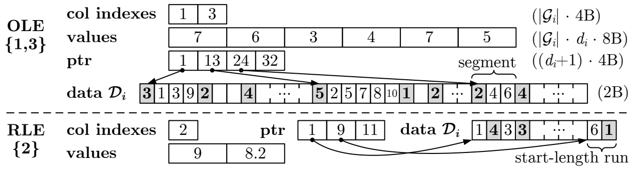


Figure 5: Data Layout OLE/RLE Column Groups.

Offset-List Encoding (OLE): Our OLE scheme divides the offset range into *segments* of fixed length $\Delta^{\text{s}} = 2^{16}$ (two bytes per offset). Each offset is mapped to its corresponding segment and encoded as the difference to the beginning of its segment. For example, the offset 155,762 lies in segment 3 $(= 1 + \lfloor(155{,}762 - 1)/\Delta^{\text{s}}\rfloor)$ and is encoded as 24,690 $(= 155{,}762 - 2\Delta^{\text{s}})$. Each segment then encodes the number of offsets with two bytes, followed by two bytes for each offset, resulting in a variable physical length in $\mathcal{D}_i$. Empty segments are represented as two bytes indicating zero length. Iterating over an OLE group entails scanning the segmented offset list and reconstructing global offsets as needed. The size S_i^{OLE} of column group $\mathcal{G}_i$ is calculated as

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

where b_{ij} denotes the number of segments of tuple $\mathbf{t}_{ij}$, $|\mathcal{O}_{ij}|$ denotes the number of offsets for $\mathbf{t}_{ij}$, and $z_i = \sum_{j=1}^{d_i} |\mathcal{O}_{ij}|$ denotes the total number of offsets in the column group. The common header has a size of $4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big)$.

Run-Length Encoding (RLE): In RLE, a sorted list of offsets is encoded as a sequence of *runs*. Each run represents a consecutive sequence of offsets, via two bytes for the starting offset and two bytes for the run length. We store starting offsets as the difference between the offset and the ending offset of the preceding run. Empty runs are used when a relative starting offset is larger than the maximum length of 2^{16}. Similarly, runs exceeding the maximum length are partitioned into smaller runs. Iterating over an RLE group entails scanning the runs and enumerating offsets per run. The size S_i^{RLE} of column group $\mathcal{G}_i$ is computed by

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

where r_{ij} is the number of runs for tuple $\mathbf{t}_{ij}$. Again, the common header has a size of $4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big)$.

3.3 Operations over Compressed Matrices

We now introduce efficient linear algebra operations over a set $\mathcal{X}$ of column groups. Matrix block operations are then composed of operations over column groups, facilitating simplicity and extensibility with regard to different encoding formats. We write $c\mathbf{v}$ to denote element-wise scalar-vector multiplication as well as $\mathbf{u} \cdot \mathbf{v}$ and $\mathbf{u} \odot \mathbf{v}$ to denote the inner and element-wise products of vectors, respectively.

Matrix-Vector Multiplication: The product $\mathbf{q} = \mathbf{X}\mathbf{v}$ of $\mathbf{X}$ and a column vector $\mathbf{v}$ can be represented with respect to column groups as $\mathbf{q} = \sum_{i=1}^{|\mathcal{X}|} \sum_{j=1}^{d_i} (\mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}) \mathbf{1}_{\mathcal{O}_{ij}}$, where $\mathbf{v}_{\mathcal{G}_i}$ is the subvector of $\mathbf{v}$ corresponding to the indexes $\mathcal{G}_i$ and $\mathbf{1}_{\mathcal{O}_{ij}}$ is the 0/1-indicator vector of offset list $\mathcal{O}_{ij}$. A straightforward way to implement this computation iterates over $\mathbf{t}_{ij}$ tuples in each group, scanning $\mathcal{O}_{ij}$ and adding $\mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ at reconstructed offsets to $\mathbf{q}$. However, pure column-wise processing would scan the $n \times 1$ output vector $\mathbf{q}$ once per tuple, resulting in cache-unfriendly behavior for the typical case of

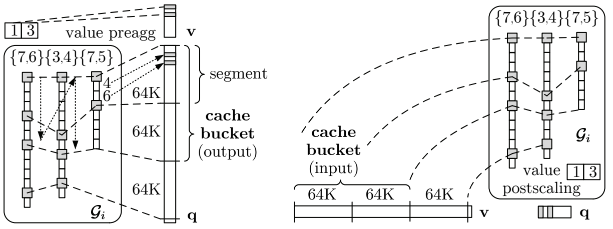
Algorithm 1 Cache-Conscious OLE Matrix-Vector

```
Input: OLE column group $\mathcal{G}_i$, vectors $\mathbf{v}$, $\mathbf{q}$, row range $[rl, ru)$
Output: Modified vector $\mathbf{q}$ (in row range $[rl, ru)$)
1: for $j$ in $[1, d_i]$ do                          // distinct tuples
2:     $\pi_{ij} \leftarrow$ skipScan($\mathcal{G}_i, j, rl$)      // find position of rl in $\mathcal{D}_i$
3:     $\mathbf{u}_{ij} \leftarrow \mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$                  // pre-aggregate value
4: for $bk$ in $[rl, ru)$ by $\Delta^c$ do       // cache buckets in [rl, ru)
5:     for $j$ in $[1, d_i]$ do                      // distinct tuples
6:         for $k$ in $[bk, \min(bk + \Delta^c, ru))$ by $\Delta^s$ do // segments
7:             if $\pi_{ij} \leq b_{ij} + |\mathcal{O}_{ij}|$ then    // physical data length
8:                 addSegment($\mathcal{G}_i, \pi_{ij}, \mathbf{u}_{ij}, k, \mathbf{q}$)    // update q, $\pi_{ij}$
```

large n. We therefore use cache-conscious schemes for OLE and RLE groups based on *horizontal, segment-aligned scans* (with benefits of up to 2.1x/5.4x for M-V/V-M in our experiments); see Algorithm 1 and Figure 6(a) for the case of OLE. Multi-threaded operations parallelize over segment-aligned partitions of rows $[rl, ru)$, which guarantees disjoint results and thus avoids partial results per thread. We find π_{ij}, the starting position of each $\mathbf{t}_{ij}$ in $\mathcal{D}_i$ via a skip scan that aggregates segment lengths until we reach rl (line 2). To minimize the overhead of finding π_{ij}, we use static scheduling (task partitioning). We further pre-compute $\mathbf{u}_{ij} = \mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ once for all tuples (line 3). For each cache-bucket of size Δ^c (such that $\Delta^c \cdot \#\text{cores} \cdot 8\,\text{B}$ fits in L3 cache, by default $\Delta^c = 2\Delta^s$), we then iterate over all distinct tuples (lines 5-8) but maintain the current positions π_{ij} as well. The inner loop (lines 6-8) then scans segments and adds $\mathbf{u}_{ij}$ via scattered writes at reconstructed offsets to the output $\mathbf{q}$ (line 8). RLE is similarly realized except for sequential writes to $\mathbf{q}$ per run, special handling of partition boundaries, and additional state for the reconstructed start offsets per tuple.

Vector-Matrix Multiplication: Column-wise compression allows for efficient vector-matrix products $\mathbf{q} = \mathbf{v}^\top \mathbf{X}$ because individual column groups update disjoint entries of the output vector $\mathbf{q}$. Each entry q_i can be expressed over columns as $q_i = \mathbf{v}^\top \mathbf{X}_{:i}$. We rewrite this multiplication in terms of a column group $\mathcal{G}_i$ as scalar-vector multiplications: $\mathbf{q}_{\mathcal{G}_i} = \sum_{j=1}^{d_i} \sum_{l \in \mathcal{O}_{ij}} v_l \mathbf{t}_{ij}$. However, a purely column-wise processing would again suffer from cache-unfriendly behavior because we scan the input vector $\mathbf{v}$ once for each distinct tuple. Our cache-conscious OLE/RLE group operations again use *horizontal, segment-aligned scans* as shown in Figure 6(b). The OLE/RLE algorithms are similar to matrix-vector but in the inner loop we sum up input-vector values according to the given offset list; finally, we scale the aggregated value once with the values in $\mathbf{t}_{ij}$. For multi-threaded operations, we parallelize over column groups, where disjoint results per column allow for simple dynamic task scheduling. The cache bucket size is equivalent to matrix-vector (by default $2\Delta^s$) except that RLE runs are allowed to cross cache bucket boundaries due to column-wise parallelization.

Special Matrix Multiplications: We also aim at *matrix-vector multiplication chains* $\mathbf{p} = \mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$, and *transpose-self matrix multiplication* $\mathbf{R} = \mathbf{X}^\top \mathbf{X}$. We effect the former via a matrix-vector multiply $\mathbf{q} = \mathbf{X}\mathbf{v}$, an uncompressed element-wise multiply $\mathbf{u} = \mathbf{q} \odot \mathbf{w}$, and a vector-matrix multiply $\mathbf{p} = (\mathbf{u}^\top \mathbf{X})^\top$ using the previously described column group operations. This block-level, composite operation scans each block twice but still avoids a second full pass over a distributed $\mathbf{X}$. Transpose-self matrix multiplication is effected via repeated vector-matrix multiplications. For each column group $\mathcal{G}_i$, we decompress $\{\, \mathbf{v}_k : k \in \mathcal{G}_i \,\}$, one column $\mathbf{v}_k$ at a time, and compute $\mathbf{p} = \mathbf{v}_k^\top \mathcal{X}_{j \geq k}$. Each non-zero $\mathbf{p}_l$ is written to both $\mathbf{R}_{k,l}$ and $\mathbf{R}_{l,k}$. Multi-threaded operations dynamically parallelize over column groups.



(a) Matrix-Vector (b) Vector-Matrix

Figure 6: Cache-Conscious OLE Operations.

Other Operations: Various common operations can be executed very efficiently over compressed matrices without scanning the offset lists. Sparse-safe matrix-scalar operations such as $\mathbf{X}^2$ or $2\mathbf{X}$ are carried out with a single pass over the set of tuples $\mathcal{T}_i$ for each column group $\mathcal{G}_i$. Append operations such as adding a column of 1's or another matrix to $\mathbf{X}$ is done via simple concatenation of column groups. Finally, unary aggregates like `sum` (or similarly `colSums`) are efficiently computed via counts by $\sum_{i=1}^{|\mathcal{X}|} \sum_{j=1}^{d_i} |\mathcal{O}_{ij}| \mathbf{t}_{ij}$. For each value, we aggregate the RLE run lengths or OLE lengths per segment, respectively. Row aggregates (e.g., `rowSums`) are computed in a cache-conscious manner.

4. COMPRESSION PLANNING

Given an uncompressed $n \times m$ matrix block $\mathbf{X}$, 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 novel sampling-based techniques for estimating the compressed size of an OLE/RLE column group $\mathcal{G}_i$. The size estimates are used for finding the initial set of compressible columns and a good column-group partitioning. Since exhaustive ($O(m^m)$) and brute-force greedy ($O(m^3)$) partitioning are infeasible, we further provide a new bin-packing-based technique that drastically reduces the number of candidate groups. Finally, we describe the overall compression algorithm including corrections for estimation errors.

4.1 Estimating Compressed Size

We present our estimators for distinct tuples d_i, non-zero tuples z_i, segments b_{ij}, and runs r_{ij} that are needed to calculate the compressed size of a column group $\mathcal{G}_i$ with formulas (1) and (2). The estimators are based on a small sample of rows $\mathcal{S}$ drawn randomly and uniformly from $\mathbf{X}$ with $|\mathcal{S}| \ll n$. In our setting, compression of an incompressible group is a worse mistake than a failure to compress a compressible group. We therefore prefer to err on the side of overestimating compressed sizes—in analogy with traditional query optimization, the key goal is to avoid disastrously bad plans.

Number of Distinct Tuples: Sampling-based estimation of the number of distinct tuples d_i is a well studied but challenging problem [14, 24, 46]. We have found that the hybrid estimator [24] is satisfactory for our purposes, compared to more expensive estimators like KMV [8] or Valiants' estimator [46]. The idea is to estimate the degree of variability in the frequencies of the tuples in $\mathcal{T}_i$ as low, medium, or high, based on the estimated squared coefficient of variation. Then we apply a "generalized jackknife" estimator that performs well for that regime to obtain an estimate $\hat{d}_i$ [24]. Such an estimator has the general 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 data-based constant, and $N^{(1)}$ is the number of tuples that appear exactly once in $\mathcal{S}$ ("singletons").

Number of OLE Segments: In general, 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 $\hat{d}_i^u = \hat{d}_i - d_i^o$, where $\hat{d}_i$ is obtained as described above. We also need to estimate the population frequencies of both 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 (see [22]), is

$$\hat{C}_i = \max\bigl(1 - N_i^{(1)}/|\mathcal{S}|, |\mathcal{S}|/n\bigr). \tag{3}$$

This estimator assumes a frequency of one for unseen tuples, computing the coverage as one minus the fraction of singletons in the sample. We add the lower sanity bound $|\mathcal{S}|/n$ to handle the 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

$$\hat{f}_{ij} = \begin{cases} (n/|\mathcal{S}|)\hat{C}_i F_{ij} & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^o \\ n(1 - \hat{C}_i)/\hat{d}_i^u & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^u. \end{cases} \tag{4}$$

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 in our experiments). There are $l = n - |\mathcal{S}|$ unobserved offsets and estimated $\hat{f}_{iq}^u = \hat{f}_{iq} - F_{iq}$ 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 kth segment ($N_{ijk} \leq l_k$). Then we estimate b_{ij} by its expected value under our maxEnt model:

$$\begin{aligned} \hat{b}_{ij} = E[b_{ij}] &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} P(N_{ijk} > 0) \\ &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} [1 - h(l_k, \hat{f}_{ij}^u, l)], \end{aligned} \tag{5}$$

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

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 (3). If $F_{i0} = 0$, then we set $\hat{f}_{i0} = 0$; this estimate maximizes $\hat{z}_i$ and hence $\hat{S}_i^{\text{OLE}}$ per our conservative estimation strategy.

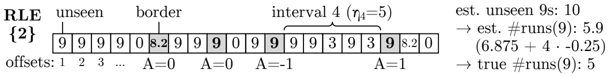


Figure 7: Estimating the Number of RLE Runs $\hat{r}_{ij}$.

Number of RLE Runs: The number of RLE runs r_{ij} for tuple $\mathbf{t}_{ij}$ is estimated as expected value of r_{ij} under the maxEnt model. Because this expected value is very hard to compute exactly and Monte Carlo approaches are too expensive, 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_{ijk}^u of unobserved $\mathbf{t}_{ij}$ instances assigned to the kth interval is hypergeometric, and we estimate f_{ijk}^u by its mean value: $\hat{f}_{ijk}^u = (\eta_k/\eta)\hat{f}_{ij}^u$, where we ignore random fluctuations about the mean. Given that $\hat{f}_{ijk}^u$ 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 [30, pp. 422-423] and we estimate r_{ijk} by its mean: $\hat{r}_{ijk} = \hat{f}_{ijk}^u(\eta_k - \hat{f}_{ijk}^u + 1)/\eta_k$. To estimate the contribution from the borders, assume that each border comprises a single observed offset. For a small sampling fraction this is the likely scenario but we handle borders of arbitrary width. If the border offset that separates intervals k and $k+1$ is an instance of $\mathbf{t}_{iq}$ for some $q \neq j$, then $A_{ijk} = 0$, where A_{ijk} is the contribution to r_{ij} from the border; in this case our estimate is simply $\hat{A}_{ijk} = 0$. If the border offset is an instance of $\mathbf{t}_{ij}$, then A_{ijk} depends on the values of the unseen offsets on either side. If both of these adjacent offsets are instances of $\mathbf{t}_{ij}$, then $A_{ijk} = -1$, because the run that spans the border has been double counted. If neither of these adjacent offsets are instances of $\mathbf{t}_{ij}$, then $A_{ijk} = 1$, because the instance of $\mathbf{t}_{ij}$ at the border constitutes a run of length 1. We estimate A_{ijk} by its approximate expected value, treating the intervals as statistically independent:

$$\begin{aligned} \hat{A}_{ijk} = E[A_{ijk}] &\approx \left(\frac{\eta_k - \hat{f}_{ijk}^u}{\eta_k}\right)\left(\frac{\eta_{k+1} - \hat{f}_{ij(k+1)}^u}{\eta_{k+1}}\right)(1) \\ &\quad + \left(\frac{\hat{f}_{ijk}^u}{\eta_k}\right)\left(\frac{\hat{f}_{ij(k+1)}^u}{\eta_{k+1}}\right)(-1) \\ &= 1 - (2\hat{f}_{ijk}^u/\eta_k) = 1 - (2\hat{f}_{ij}^u/\eta). \end{aligned} \tag{6}$$

We modify this formula appropriately for borders at the first or last offset. Our final estimate is $\hat{r}_{ij} = \sum_k \hat{r}_{ijk} + \sum_k \hat{A}_{ijk}$.

4.2 Partitioning Columns into Groups

A greedy brute-force method for partitioning a set of compressible columns into groups starts with singleton groups and executes merging iterations. At each iteration, we merge the two groups having maximum compression ratio (sum of their compressed sizes divided by the compressed size of the merged group). We terminate when no further space reductions are possible, i.e., no compression ratio exceeds 1. Although compression ratios are estimated from a sample, the cost of the brute-force scheme is $O(m^3)$, which is infeasible.

Bin Packing: We observed empirically that the brute-force method usually generates groups of no more than five

Algorithm 2 Matrix Block Compression

```
Input: Matrix block $\mathbf{X}$ of size $n \times m$
Output: A set of compressed column groups $\mathcal{X}$
 1: $C^{\mathrm{C}} \leftarrow \emptyset,\ C^{\mathrm{UC}} \leftarrow \emptyset,\ \mathcal{G} \leftarrow \emptyset,\ \mathcal{X} \leftarrow \emptyset$
 2: // Planning phase – – – – – – – – – – – – – – – – – – –
 3: $\mathcal{S} \leftarrow$ sampleRowsUniform($\mathbf{X}$, sample_size)
 4: for all column $k$ in $\mathbf{X}$ do                    // classify
 5:     cmp_ratio $\leftarrow \hat{z}_i\alpha / \min(\hat{S}_k^{\mathrm{RLE}}, \hat{S}_k^{\mathrm{OLE}})$
 6:     if cmp_ratio $> 1$ then
 7:         $C^{\mathrm{C}} \leftarrow C^{\mathrm{C}} \cup k$
 8:     else
 9:         $C^{\mathrm{UC}} \leftarrow C^{\mathrm{UC}} \cup k$
10: bins $\leftarrow$ runBinPacking($C^{\mathrm{C}}$)                   // group
11: for all bin $b$ in bins do
12:     $\mathcal{G} \leftarrow \mathcal{G} \cup$ groupBruteForce($b$)
13: // Compression phase – – – – – – – – – – – – – – – – –
14: for all column group $\mathcal{G}_i$ 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:         $C^{\mathrm{UC}} \leftarrow C^{\mathrm{UC}} \cup k$
22:     while $|\mathcal{G}_i| > 0$
23: return $\mathcal{X} \leftarrow \mathcal{X} \cup$ createUCGroup($C^{\mathrm{UC}}$)
```

columns. Further, we noticed that the time needed to estimate a group size increases as the sample size, the number of distinct tuples, or the matrix density increases. These two observations motivate a heuristic strategy where we partition the columns into a set of small *bins* and then apply the brute-force method within each bin to form the column groups. We use a bin-packing algorithm to assign columns to bins. The weight of each column indicates its estimated contribution to the overall runtime of the brute-force partitioning. The capacity of a bin is chosen to ensure moderate brute-force runtime per bin. Intuitively, bin packing minimizes the number of bins, which should maximize the number of columns within each bin and hence grouping potential, while controlling the processing costs.

Bin Weights: We set each bin capacity to $w = \beta\gamma$ and the weight of the i^{th} column to $\hat{d}_i/n$ (i.e., the ratio of distinct tuples to rows), where β is a tuning parameter and γ also constrains grouping decisions: $\hat{d}_i/n \leq \gamma$. If this constraint does not hold, we consider column i as ineligible for grouping to prune high cardinality columns. We made the design choice of a constant bin capacity—independent of the number of non-zeros—to ensure constant compression ratios and throughput irrespective of blocking configurations. We use the first-fit heuristic to solve the bin-packing problem.

4.3 Compression Algorithm

We now describe the overall algorithm for creating compressed matrix blocks (Algorithm 2). Note that we transpose the input in case of row-major dense or sparse formats to avoid performance issues due to column-wise processing.

Planning Phase (lines 2-12)**:** Planning starts by drawing a sample of rows from $\mathbf{X}$. For each column i, the sample is first used to estimate the compressed column size S_i^{C} by $\hat{S}_i^{\mathrm{C}} = \min(\hat{S}_i^{\mathrm{RLE}}, \hat{S}_i^{\mathrm{OLE}})$, where $\hat{S}_i^{\mathrm{RLE}}$ and $\hat{S}_i^{\mathrm{OLE}}$ are obtained by substituting the estimated $\hat{d}_i$, $\hat{z}_i$, $\hat{r}_{ij}$, and $\hat{b}_{ij}$ into formulas (1) and (2). We conservatively estimate the uncompressed column size as $\hat{S}_i^{\mathrm{UC}} = \hat{z}_i\alpha$, which covers both dense and sparse with moderate underestimation and allows column-wise decisions independent of $|C^{\mathrm{UC}}|$ (where sparse-row overheads might be amortized in case of many columns). Columns whose estimated compression ratio $\hat{S}_i^{\mathrm{UC}}/\hat{S}_i^{\mathrm{C}}$ exceed 1 are added to a compressible set C^{C}. In a last step, we divide the columns in C^{C} into bins and apply the greedy brute-force algorithm within each bin to form column groups.

Compression Phase (lines 13-23)**:** The compression phase first obtains exact information about the parameters of each column group and uses this information in order to adjust the groups, correcting for any errors induced by sampling during planning. The exact information is also used to make the final decision on encoding formats for each group. In detail, 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 together with the uncompressed lists of offsets for the tuples. The big lists for all of the column 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 additional cost. These parameters are used in turn to calculate the exact compressed sizes S_i^{RLE} and S_i^{OLE} and exact compression ratio $S_i^{\mathrm{UC}}/S_i^{\mathrm{C}}$ 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, i.e., purportedly compressible groups that are in fact incompressible. Instead of simply storing false-positive OLE/RLE groups as UC group, we attempt to correct the group by removing the column with largest estimated compressed size. The correction process is repeated until the remaining group is either compressible or empty. After each group has been corrected, we choose the optimal encoding scheme for each compressible group $\mathcal{G}_i$ using the exact parameter values d_i, z_i, b_{ij}, and r_{ij} together with the formulas (1) and (2). The incompressible columns are collected into a single UC column group.

5. EXPERIMENTS

We study CLA in SystemML over a variety of ML programs and real-world datasets. The major insights are:

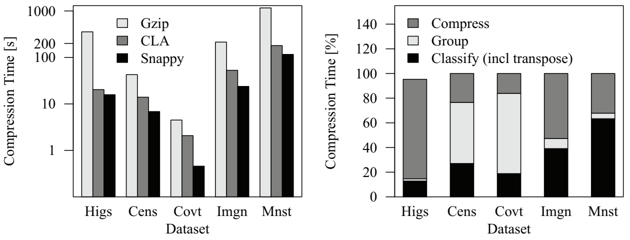
Operations Performance: CLA achieves in-memory matrix-vector multiply performance close to uncompressed. Sparse-safe scalar and aggregate operations show huge improvements due to value-based computation.

Compression Ratio: CLA yields substantially better compression ratios than lightweight general-purpose compression. Hence, CLA provides large end-to-end performance improvements, of up to 26x, when uncompressed or lightweight-compressed matrices do not fit in memory.

Effective Compression Planning: Sampling-based compression planning yields both reasonable compression time and good compression plans, i.e., good choices of encoding formats and co-coding schemes. We thus obtain good compression ratios at costs that are easily amortized.

5.1 Experimental Setting

Cluster Setup: We ran all experiments on a 1+6 node cluster, i.e., one head node of 2x4 Intel E5530 @ 2.40 GHz-2.66 GHz with hyper-threading and 64 GB RAM @800 MHz, as well as 6 nodes of 2x6 Intel E5-2440 @ 2.40 GHz-2.90 GHz with hyper-threading, 96 GB RAM @1.33 GHz (ECC, registered), 12x2 TB disks, 10Gb Ethernet, and Red Hat Enterprise Linux Server 6.5. The nominal peak performance per node for memory bandwidth and floating point operations are 2x32 GB/s from local memory (we measured



(a) Absolute Time (b) Time Breakdown

Figure 8: Compression Time.

47.9 GB/s), 2x12.8 GB/s over QPI (Quick Path Interconnect), and 2x115.2 GFLOP/s. We used OpenJDK 1.7.0_95, and Apache Hadoop 2.2.0, configured with 11 disks for HDFS and local working directories. We ran Apache Spark 1.4, in yarn-client mode, with 6 executors, 25 GB driver memory, 60 GB executor memory, and 24 cores per executor. Finally, we used Apache SystemML 0.9 with default configuration, except for a larger block size of $16K$ rows.

Implementation Details: We integrated CLA into SystemML; if enabled, the system automatically injects—for any multi-column input matrix—a so-called `compress` operator via new rewrites, after initial read or text conversion but before checkpoints. This applies to both single-node and distributed Spark operations, where the execution type is chosen based on memory estimates. The `compress` operator transforms an uncompressed into a compressed matrix including compression planning. For distributed Spark operations, we compress individual matrix blocks independently. Making our compressed matrix block a subclass of the uncompressed matrix block yielded seamless integration of all operations, serialization, and buffer pool interactions.

ML Programs and Datasets: For end-to-end experiments, we used several common algorithms: LinregCG (linear regression conjugate gradient), LinregDS (linear regression direct solve), MLogreg (multinomial logistic regression), GLM (generalized linear models, poisson log), L2SVM (L2 regularized support vector machines), and PCA (principal component analysis) as described in Table 2. We configured these algorithms as follows: max outer iterations $moi = 10$, max inner iterations $mii = 5$, intercept $icp = 0$, convergence tolerance $\epsilon = 10^{-9}$, regularization $\lambda = 10^{-3}$. Note that LinregDS/PCA are non-iterative and LinregCG is the only iterative algorithm without nested loops. We ran all experiments over real-world and scaled real-world datasets, introduced in Table 1. For our large-scale experiments, we used (1) the *InfiMNIST* data generator [13] to create an Mnist480m dataset of 480 million observations with 784 features and binomial class labels (1.1 TB in binary format), as well as (2) replicated versions of the ImageNet dataset.

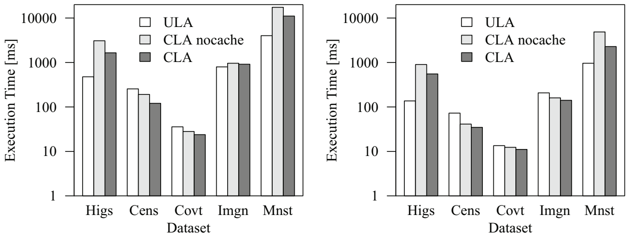
Baseline Comparisons: In order to isolate the effects of compression, we compare against Apache SystemML 0.9 (Feb 2016) with (1) uncompressed linear algebra (ULA), (2) heavyweight compression: Gzip, and (3) lightweight compression: Snappy, where we use native compression libraries and ULA. Finally, we also compare with (4) CSR-VI [34], a sparse format with dictionary encoding.

5.2 Compression and Operations

To provide a deeper understanding of both compression and operations performance, we discuss several micro benchmarks. Recall that our overall goal is to achieve excellent compression while maintaining operations performance close to other methods, in order to achieve significant end-to-end performance benefits. We conducted these micro benchmarks on a single worker node with 80 GB Java heap size. We used 5 warmup runs for just-in-time compilation, and report the average execution time over 20 subsequent runs.

Table 3: Compression Plans of Individual Datasets.

Dataset	m	$\lvert\mathcal{X}\rvert$	#OLE	#RLE	#UC	#Vals
Higgs	28	17	16	0	1	218,738
Census	68	11	11	0	0	40,202
		13	13	0	0	56,413
Covtype	54	35	10	25	0	15,957
		43	10	33	0	15,957
ImageNet	900	502	502	0	0	159,161
		510	510	0	0	167,123
Mnist8m	784	709	610	94	0	308,544
		724	625	111	0	387,425



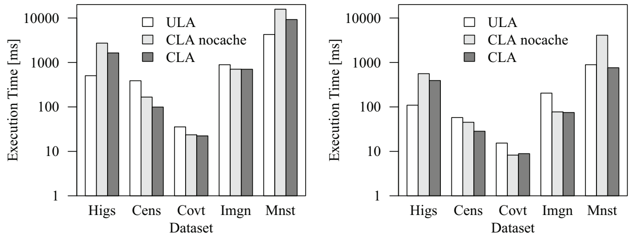
(a) Single-Threaded (b) Multi-Threaded

Figure 9: Matrix-Vector Multiplication Time.

Compression: Figure 8 shows the time of creating a single compressed matrix block. Figure 8(a) shows the absolute compression times in log scale, where we see reasonable performance with an average bandwidth across all datasets of roughly 100 MB/s, single-threaded. In comparison, the single-threaded compression throughput of the general-purpose Gzip and Snappy, ranged from 6.9 MB/s to 35.6 MB/s and 156.8 MB/s to 353 MB/s, respectively. Figure 8(b) further shows the time breakdown of individual compression steps, where our planning phase comprises both classification and grouping. Bar heights below 100% are due to the final extraction of uncompressed column groups. Depending on the dataset, any of the three compression steps (sampling-based classification and column grouping, or the offset list extraction) can turn into bottlenecks.

Summary of Compression Plans: As a precondition for understanding the micro benchmarks on operations performance, we first summarize the compression layouts for our datasets. Due to sample-based compression planning, there are moderate variations between layouts for different runs. However, the differences of compressed sizes were less than 2.5% in all cases. Table 3 shows the layouts observed over 20 runs, where we report min and max counts as two rows if they show differences. We see that (1) OLE is more common than RLE, (2) Higgs is the only dataset with an uncompressed column group, (3) co-coding was applied on all datasets, and (4) Higgs, Mnist8m and to some extent ImageNet, show a large number of values, although we already excluded the uncompressed column group of Higgs.

Matrix-Vector Multiplication: Figure 9(a) and 9(b) show the single- and multi-threaded matrix-vector multiplication time. Despite row-wise updates of the target vector (in favor of uncompressed row-major layout), CLA shows performance close to ULA, with two exceptions of Higgs and Mnist8m, where CLA performs significantly worse. This behavior is mostly caused by (1) a large number of values which require multiple passes over the output vector, and



(a) Single-Threaded (b) Multi-Threaded

Figure 10: Vector-Matrix Multiplication Time.

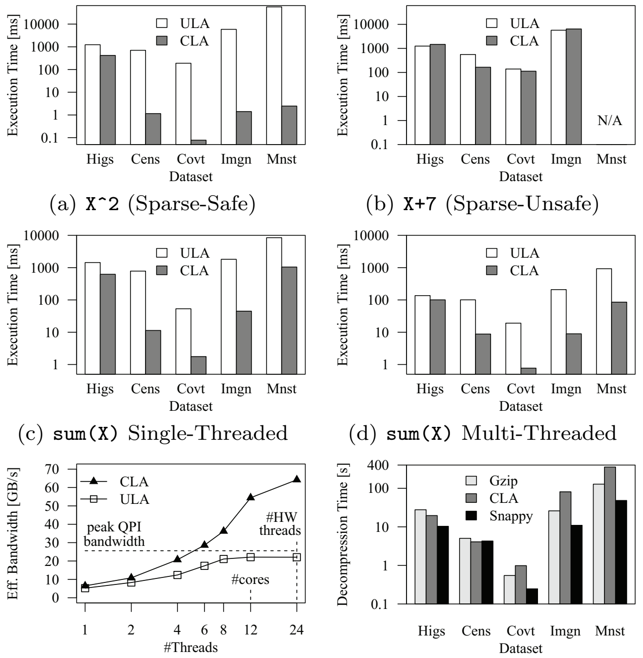
(2) the size of the output vector. For Higgs (11M rows) and Mnist8m (8M rows), the target vector does not entirely fit into the L3 cache (15 MB). Accordingly, we see substantial improvements by cache-conscious CLA operations, especially for multi-threaded due to cache thrashing effects. Multi-threaded operations show a speedup similar to ULA due to parallelization over logical row partitions, in some cases even better as explained later. ULA constitutes a competitive baseline because its multi-threaded implementation achieves peak remote memory bandwidth of ≈ 25 GB/s.

Vector-Matrix Multiplication: Figures 10(a) and 10(b) further show the single- and multi-threaded vector-matrix multiplication time. The column-wise updates favor CLA's column-wise layout and hence we see generally better performance. CLA is again slower for Higgs and Mnist8m, due to large input vectors that exceed the L3 cache size as well as repeated scattered scans of these vectors for many values. However, cache-conscious CLA operations mitigate this effect almost entirely. ULA is again a strong baseline at peak remote memory bandwidth. Interestingly, multi-threaded CLA operations show a better speedup because ULA becomes memory-bandwidth bound, whereas CLA has less bandwidth requirements due to smaller compressed size, and multi-threading mitigates additional overheads. For example, Figure 11(e) shows the effective bandwidth on ImageNet, with varying number of threads, where CLA exceeds the peak remote memory bandwidth of 25 GB/s by 2.5x.

Matrix-Scalar Operations: We also investigate sparse-safe and -unsafe matrix-scalar operations, where the former only processes non-zero values. Figure 11(a) shows results for the sparse-safe `X^2`. CLA performs `X^2` on the distinct value tuples with a shallow (by-reference) copy of existing offset lists, whereas ULA has to compute every non-zero entry. We see improvements of three to four orders of magnitude, except for Higgs which contains a large uncompressed group. Figure 11(b) shows the results for the sparse-unsafe `X+7`, where CLA and ULA perform similar because CLA has to materialize modified offset lists that include added and removed values. For Census, we see better performance due to very small offset lists. Finally, Mnist8m is not applicable here because dense matrix blocks are limited to 16 GB.

Unary Aggregate Operations: Figures 11(c) and 11(d) compare the single- and multi-threaded aggregation time for `sum(X)`. Due to efficient counting per value—via scanning of OLE segment lengths and RLE run lengths—we see improvements of one to two orders of magnitude compared to ULA (at peak memory bandwidth). The only exception is again Higgs due its uncompressed column group.

Decompression: In the rare case of unsupported operations, we decompress and perform ULA operations. Figure 11(f) shows the decompression time. In contrast, Gzip or Snappy need to decompress block-wise for *every* operation.



(a) `X^2` (Sparse-Safe) (b) `X+7` (Sparse-Unsafe)

(c) `sum(X)` Single-Threaded (d) `sum(X)` Multi-Threaded

(e) Vector-Matrix Bandwidth (f) Decompression Time

Figure 11: Additional Detailed Micro Benchmarks.

5.3 Comparison to CSR-VI

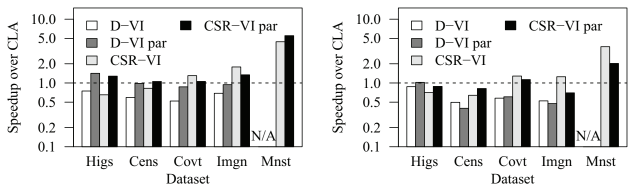
We are aware of a single existing work on *lossless* matrix value compression: CSR-VI (CSR Value Indexed) [34], via dictionary encoding. Our comparison includes CSR-VI and a derived dense format that we call D-VI, both of which use custom implementations for 1, 2, and 4-byte codes.

Compression Ratio: Table 4 shows the compression ratio of CSR-VI, D-VI, and CLA compared to uncompressed matrix blocks in Modified-CSR (additional header per row) or dense format. We see that CLA achieves substantial size improvements for compressible sparse *and* dense datasets. The compression potential for CSR-VI and D-VI is determined by the given number of distinct values.

Operations Performance: Figure 12 further shows the single- and multi-threaded (par) matrix-vector and vector-matrix multiplication performance of CSR-VI and D-VI, normalized to CLA, where a speedup > 1 indicates improvements over CLA. We see that CSR-VI and D-VI achieve performance close to CLA for matrix-vector because it favors row-major formats, while for vector-matrix CLA performs

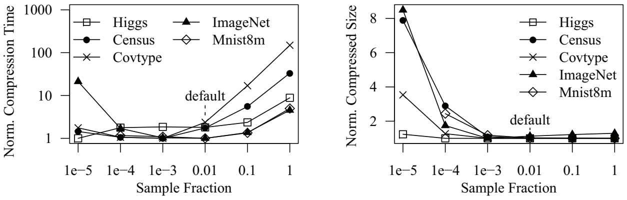
Table 4: Compression Ratio CSR-VI vs. CLA.

Dataset	Sparse	#Values	CSR-VI	D-VI	CLA
Higgs	N	8,083,944	1.04	1.90	**2.03**
Census	N	46	3.62	7.99	**27.46**
Covtype	Y	6,682	3.56	2.48	**12.73**
ImageNet	Y	824	2.07	1.93	**7.38**
Mnist8m	Y	255	2.53	N/A	**6.14**



(a) Matrix-Vector (b) Vector-Matrix

Figure 12: Performance CSR-VI vs. CLA.



(a) Compression Time (b) Compressed Size

Figure 13: Influence of Sample Fraction.

generally better. Mnist8m is an exception where CSR-VI always performs better due to 1-byte value encoding, whereas CLA uses many value offset lists as shown in Table 3.

Overall, CLA shows similar operations performance at significantly better compression ratios which is crucial for end-to-end performance improvements of large-scale ML.

5.4 Parameter Influence and Accuracy

We now evaluate CLA parameter sensitivity and size estimation accuracy, where we report the average of 5 runs.

Sample Fraction: Sampling-based compression planning is crucial for fast compression. Figure 13 shows the compression time and compressed size (min-normalized) with varying sample fraction. We see huge time improvements using small fractions. However, very small fractions cause—due to estimation errors—increasing sizes, which also impact compression time. Sampling is especially important for Census and Covtype, where we spend a substantial fraction of time on column grouping (compare Figure 8(b)). By default, we use a conservative, low-risk sample fraction of 0.01.

Bin Weights: Our bin-packing-based column group partitioning reduces the number of candidate column groups with bin capacity $w = \beta\gamma$. The larger β the smaller the number of groups, but the higher the execution time, especially, for datasets with many columns like ImageNet or Mnist8m. Our default of $\beta = 0.05$ achieves a good tradeoff between compression time and compressed size, and yielded compressed sizes within 23.6% of the observed minimum.

Estimation Accuracy: We compare our CLA size estimators with a systematic excerpt [17] (first $0.01n$ rows), that allows to observe compression ratios. Table 5 reports the ARE (absolute ratio error) $|\hat{S} - S|/S$ of estimated size $\hat{S}$ (before corrections) to actual CLA compressed size S. CLA shows significantly better accuracy due to robustness against skew and effects of value tuples. Datasets with RLE groups (Covtype, Mnist8m) show generally higher errors since RLE is difficult to predict. Excerpt also resulted in worse plans because column grouping mistakes could not be corrected.

Table 5: Size Estimation Accuracy (Average ARE).

Dataset	Higgs	Census	Covtype	ImageNet	Mnist8m
Excerpt [17]	28.8%	173.8%	111.2%	24.6%	**12.1%**
CLA Est.	**16.0%**	**13.2%**	**56.6%**	**0.6%**	39.4%

Conservative default parameters together with compression corrections and fallbacks for incompressible columns led to a robust design *without* the need for tuning per dataset.

5.5 End-to-End Experiments

To study end-to-end CLA benefits, we ran several algorithms over subsets of Mnist480m and ImageNet. We report end-to-end runtime (average of 3 runs), including read from HDFS, Spark context creation, and compression. The baselines are ULA and Spark's RDD compression with Snappy.

Table 6: Mnist8m Deserialized 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**
CLA	9.9 GB	8.4 GB	6 GB	4.4 GB	**3.6 GB**

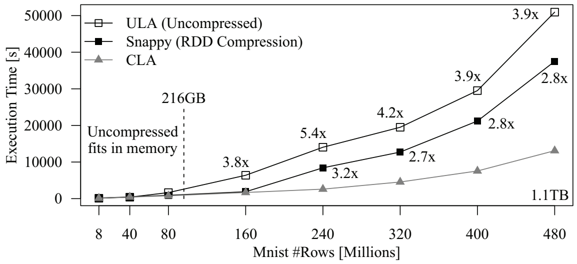


Figure 14: L2SVM End-to-End Performance Mnist.

RDD Storage: ULA and CLA use the deserialized storage level `MEM_AND_DISK`, while Snappy uses `MEM_AND_DISK_SER` because RDD compression requires serialized data. ULA also uses `MEM_AND_DISK_SER` for sparse matrices whose sizes exceed aggregated memory. Table 6 shows the RDD storage size of Mnist8m with varying SystemML block size. For $16K$, we observe a compression ratio of 2.5x for Snappy but 5x for CLA. We obtained similar ratios for larger Mnist subsets.

L2SVM on Mnist: We first investigate L2SVM as a common classification algorithm. Given the described setup, we have an aggregated memory of $6 \cdot 60\,\text{GB} \cdot 0.6 = 216\,\text{GB}$. SystemML uses hybrid runtime plans, where only operations that exceed the driver memory are executed as distributed Spark instructions; all other vector operations are executed—similarly for all baselines—as single-node operations at the driver. For L2SVM, we have two scans of $\mathbf{X}$ per outer iteration (matrix-vector and vector-matrix), whereas all inner-loop operations are purely single-node for the data at hand. Figure 14 shows the results. In comparison to our goals from Figure 1, Spark spills data to disk at granularity of partitions (128 MB as read from HDFS), leading to a graceful performance degradation. As long as the data fits in aggregated memory (Mnist80m, 180 GB), all runtimes are almost identical, with Snappy and CLA showing overheads of up to 25% and 10%, respectively. However, if the ULA format no longer fits in aggregated memory (Mnist160m, 360 GB), we see significant improvements from compression because the size reduction avoids spilling, i.e., reads per iteration. The larger compression ratio of CLA allows to fit larger datasets into memory (e.g., Mnist240m). Once even the CLA format no longer fits in memory, the runtime differences converge to the differences in compression ratios.

Other ML Algorithms on Mnist: Finally, we study 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). Table 7 shows the results for the interesting points of Mnist40m (90 GB), where all datasets fit in memory, and Mnist240m (540 GB), where neither uncompressed nor Snappy-compressed datasets entirely fit in memory. For Mnist40m and iterative algorithms, we see similar ULA/CLA performance but a 50% slowdown with Snappy. This is because RDD compression incurs decompression overhead per iteration, whereas CLA's initial compression cost is amortized over multiple iterations. For non-iterative algorithms, CLA is up to 32% slower while Snappy shows less than 12% overhead. Beside the initial compression overhead, CLA also shows less efficient TSMM

Table 7: End-to-End Performance Mnist40m/240m.

Algorithm	**Mnist40m** (90 GB)			**Mnist240m** (540 GB)		
	ULA	Snappy	CLA	ULA	Snappy	CLA
Mlogreg	630 s	875 s	**622 s**	83,153 s	27,626 s	**4,379 s**
GLM	409 s	647 s	**397 s**	74,301 s	23,717 s	**2,787 s**
LinregCG	**173 s**	220 s	176 s	2,959 s	1,493 s	**902 s**
LinregDS	**187 s**	208 s	247 s	1,984 s	1,444 s	**1,305 s**
PCA	**186 s**	203 s	242 s	1,203 s	**1,020 s**	1,287 s

Table 8: End-to-End Performance ImageNet15/150.

Algorithm	**ImageNet15** (65 GB)			**ImageNet150** (650 GB)		
	ULA	Snappy	CLA	ULA	Snappy	CLA
L2SVM	**157 s**	199 s	159 s	25,572 s	8,993 s	**3,097 s**
Mlogreg	255 s	400 s	**250 s**	100,387 s	31,326 s	**4,190 s**
GLM	190 s	304 s	**186 s**	60,363 s	16,002 s	**2,453 s**
LinregCG	**69 s**	98 s	71 s	3,829 s	997 s	**623 s**
LinregDS	207 s	216 s	**118 s**	3,648 s	2,720 s	**1,154 s**
PCA	211 s	215 s	**119 s**	2,765 s	2,431 s	**1,107 s**

performance, while the RDD decompression overhead, is mitigated by initial read costs. For Mnist240m, we see significant performance improvements by CLA—of up to 26x and 8x—compared to ULA and RDD compression for Mlogreg and GLM. This is due to many inner iterations with RDD operations in the outer and inner loop. In contrast, for LinregCG, we see only moderate improvements due to a single loop with one matrix-vector chain per iteration, where the CLA runtime was dominated by initial read and compression. Finally, for LinregDS, CLA shows again slightly inferior TSMM performance but moderate improvements compared to ULA. Overall CLA shows positive results with significant improvements for iterative algorithms due to smaller memory bandwidth requirements and reduced I/O.

ML Algorithms on ImageNet: To validate the end-to-end results, we study the same algorithms over replicated ImageNet datasets. Due to block-wise compression, replication did not affect the compression ratio. Table 8 shows the results for ImangeNet15 (65 GB) that fits in memory, and ImageNet150 (650 GB). For LinregDS and PCA, CLA performs better than on Mnist due to superior vector-matrix and thus TSMM performance (see Figure 10). Overall, we see similar results with improvements of up to 24x and 7x.

6. RELATED WORK

We generalize sparse matrix representations via compression and accordingly review related work of database compression, sparse linear algebra, and compression planning.

Compressed Databases: The notion of compressing databases appears in the literature back in the early 1980s [5, 18], although most early work focuses on the use of general-purpose techniques like Huffman coding. An important exception is the Model 204 database system, which used compressed bitmap indexes to speed up query processing [38]. More recent systems that use bitmap-based compression include FastBit [49], Oracle [39], and Sybase IQ [45]. Graefe and Shapiro's 1991 paper "Data Compression and Database Performance" more broadly introduced the idea of compression to improve query performance by evaluating queries in the compressed domain [23], primarily with dictionary-based compression. Westmann et al. explored storage, query processing and optimization with regard to lightweight compression techniques [47]. Later, Raman and Swart investigated query processing over heavyweight Huffman coding schemes [40], where they have also shown the benefit of column co-coding. Recent examples of relational database systems that use multiple types of compression to speed up query processing include C-Store/Vertica [43], SAP HANA [11], IBM DB2 with BLU Acceleration [41], Microsoft SQL Server [36], and HyPer [35]. SciDB—as an array database—also uses compression but decompressed arrays block-wise for each operation [44]. Further, Kimura et al. made a case for compression-aware physical design tuning to overcome suboptimal design choices [33], which requires to estimate sizes of compressed indexes. Existing estimators focus on compression schemes such as null suppression and dictionary encoding [29], where the latter is again related to estimating the number of distinct values. Other estimators focus on index layouts such as RID list and prefix key compression [9].

Sparse Matrix Representations: Sparse matrix formats have been studied intensively. Common formats include CSR (compressed sparse rows), CSC (compressed sparse columns), COO (coordinate), DIA (diagonal), ELL (ellpack-itpack generalized diagonal), and BSR (block sparse row) [42]. These formats share the characteristic of encoding non-zero values along with their positions. Examples of hybrid formats—that try to combine advantages—are HYB (hybrid format) [6] that splits a matrix into ELL and COO areas to mitigate irregular structures, and SLACID [32] that represents matrices in CSR format with COO deltas for a seamless integration with SAP HANA's delta architecture. Especially for sparse matrix-vector multiplication on GPUs there are also operation and architecture-aware formats like BRC (blocked row-column format) [3] that applies rearrangement of rows by number of non-zeros and padding. Williams et al. studied various optimizations and storage formats for sparse matrix-vector multiplications on multi-core systems [48]. Finally, Kourtis et al. already introduced compression techniques for sparse matrix formats, where they applied run-length encoding of column index deltas [31, 34] and dictionary encoding [34]. In contrast to existing work, we aim at sparse and dense column value compression with heterogeneous encodings and co-coding.

Compression Planning: The literature for compression and deduplication planning is relatively sparse and focuses on a priori estimation of compression ratios for heavyweight algorithms on generic data. A common strategy [17] is to experimentally compress a small segment of the data (excerpt) and observe the compression ratio. The drawbacks to this approach [26] are that (1) the segment may not be representative of the whole dataset and (2) the compression step can be very expensive because the runtime of many algorithms varies inversely with the achieved compression. The authors in [25] propose a procedure for estimating deduplication compression ratios in large datasets, but the algorithm requires a complete pass over the data. The first purely sampling-based approach to compression estimation is presented in [26] in the context of Huffman coding of generic data. The idea is to sample different locations in the data file and compute "local" compression ratios. These local estimates are treated as independent and averaged to yield an overall estimate together with probabilistic error bounds. This technique does not readily extend to our setting because our OLE and RLE methods do not have the required "bounded locality" properties, which assert that the compressibility of a given byte depends on a small number of nearby bytes. Overall, in contrast to prior work, we propose a method for estimating the compression when several specific lightweight methods are applied to numeric matrices.

7. CONCLUSIONS

We have initiated work on compressed linear algebra (CLA), in which matrices are compressed with lightweight techniques and linear algebra operations are performed directly over the compressed representation. We introduced effective column encoding schemes, efficient operations over compressed matrices, and an efficient sampling-based compression algorithm. Our experiments show operations performance close to the uncompressed case and compression ratios similar to heavyweight formats like Gzip but better than lightweight formats like Snappy, providing significant performance benefits when data does not fit into memory. Thus, we have demonstrated the general feasibility of CLA, enabled by declarative ML that hides the underlying physical data representation. CLA generalizes sparse matrix representations, encoding both dense and sparse matrices in a universal compressed form. CLA is also broadly applicable to any system that provides blocked matrix representations, linear algebra, and physical data independence. Interesting future work includes (1) full optimizer integration, (2) global planning and physical design tuning, (3) alternative compression schemes, and (4) operations beyond matrix-vector.

8. REFERENCES

[1] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, 2016.

[2] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB J.*, 23(6), 2014.

[3] A. Ashari et al. An Efficient Two-Dimensional Blocking Strategy for Sparse Matrix-Vector Multiplication on GPUs. In *ICS (Intl. Conf. on Supercomputing)*, 2014.

[4] A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP (Principles and Practice of Parallel Programming)*, 2015.

[5] M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. *TSE (Trans. SW Eng.)*, 11(10), 1985.

[6] N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC (Supercomputing Conf.)*, 2009.

[7] J. Bergstra et al. Theano: a CPU and GPU Math Expression Compiler. In *SciPy*, 2010.

[8] K. S. Beyer et al. On Synopses for Distinct-Value Estimation Under Multiset Operations. In *SIGMOD*, 2007.

[9] B. Bhattacharjee et al. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2), 2009.

[10] S. Bhattacherjee et al. PStore: An Efficient Storage Framework for Managing Scientific Data. In *SSDBM*, 2014.

[11] C. Binnig et al. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*, 2009.

[12] M. Boehm et al. Declarative Machine Learning – A Classification of Basic Properties and Types. *CoRR*, 2016.

[13] L. Bottou. The infinite MNIST dataset. `http://leon.bottou.org/projects/infimnist`.

[14] M. Charikar et al. Towards Estimation Error Guarantees for Distinct Values. In *SIGMOD*, 2000.

[15] R. Chitta et al. Approximate Kernel k-means: Solution to Large Scale Kernel Clustering. In *KDD*, 2011.

[16] J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[17] C. Constantinescu and M. Lu. Quick Estimation of Data Compression and De-duplication for Large Storage Systems. In *CCP (Data Compression, Comm. and Process.)*, 2011.

[18] G. V. Cormack. Data Compression on a Database System. *Commun. ACM*, 28(12), 1985.

[19] S. Das et al. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[21] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[22] I. J. Good. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika*, 1953.

[23] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *Applied Computing*, 1991.

[24] P. J. Haas and L. Stokes. Estimating the Number of Classes in a Finite Population. *J. Amer. Statist. Assoc.*, 93(444), 1998.

[25] D. Harnik et al. Estimation of Deduplication Ratios in Large Data Sets. In *MSST (Mass Storage Sys. Tech.)*, 2012.

[26] D. Harnik et al. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. In *FAST*, 2013.

[27] B. Huang et al. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[28] B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.

[29] S. Idreos et al. Estimating the Compression Fraction of an Index using Sampling. In *ICDE*, 2010.

[30] N. L. Johnson et al. *Univariate Discrete Distributions.* Wiley, New York, 2nd edition, 1992.

[31] V. Karakasis et al. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. *TPDS (Trans. Par. and Dist. Systems)*, 24(10), 2013.

[32] D. Kernert et al. SLACID - Sparse Linear Algebra in a Column-Oriented In-Memory Database System. In *SSDBM*, 2014.

[33] H. Kimura et al. Compression Aware Physical Database Design. *PVLDB*, 4(10), 2011.

[34] K. Kourtis et al. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *CF (Computing Frontiers)*, 2008.

[35] H. Lang et al. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, 2016.

[36] P. Larson et al. SQL Server Column Store Indexes. In *SIGMOD*, 2011.

[37] M. Lichman. UCI Machine Learning Repository: Higgs, Covertype, US Census (1990). `archive.ics.uci.edu/ml/`.

[38] P. E. O'Neil. Model 204 Architecture and Performance. In *High Performance Transaction Systems*. 1989.

[39] Oracle. *Data Warehousing Guide, 11g Release 1*, 2007.

[40] V. Raman and G. Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*, 2006.

[41] V. Raman et al. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 6(11), 2013.

[42] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994.

[43] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[44] M. Stonebraker et al. The Architecture of SciDB. In *SSDBM*, 2011.

[45] Sysbase. *IQ 15.4 System Administration Guide*, 2013.

[46] G. Valiant and P. Valiant. Estimating the Unseen: An n/log(n)-sample Estimator for Entropy and Support Size, Shown Optimal via New CLTs. In *STOC*, 2011.

[47] T. Westmann et al. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3), 2000.

[48] S. Williams et al. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC (Supercomputing Conf.)*, 2007.

[49] K. Wu et al. Optimizing Bitmap Indices With Efficient Compression. *TODS*, 31(1), 2006.

[50] L. Yu et al. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, 2015.

[51] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[52] C. Zhang et al. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.