Scaling Machine Learning via Compressed Linear Algebra

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 and enable very fast matrix-vector operations on in-memory data. General-purpose, heavy- and lightweight compression techniques struggle to achieve both good compression ratios and fast decompression speed to enable block-wise uncompressed operations. Compressed linear algebra (CLA) avoids these problems by applying lightweight lossless database compression techniques to matrices and then executing linear algebra operations such as matrix-vector multiplication directly on the compressed representations. The key ingredients are effective column compression schemes, cache-conscious operations, and an efficient sampling-based compression algorithm. Experiments on an initial implementation in SystemML show in-memory operations performance close to the uncompressed case and good compression ratios. We thereby obtain significant end-to-end performance improvements up to 26x or reduced memory requirements.

1. INTRODUCTION

Large-scale machine learning (ML) leverages large data collections in order to find interesting patterns and build robust predictive models [9, 10]. Applications include regression analysis, classification, and recommendations. Data-parallel frameworks such as MapReduce [11], Spark [22], and Flink [2] are often used for cost-effective parallelized model building on commodity hardware.

Declarative ML: State-of-the-art, large-scale ML systems support declarative ML algorithms [5], expressed in high-level languages, that comprise linear algebra operations, i.e., matrix multiplications, aggregations, element-wise and statistical computations. Examples—at varying abstraction levels—are SystemML [6], SciDB [20], Cumulon [15], DMac [21], and TensorFlow [1]. A high level of abstraction gives data scientists the flexibility to create and customize ML algorithms without worrying about data and cluster characteristics, underlying data representations (e.g., sparse/dense format) or execution plan generation. We aim to improve the performance of declarative ML algorithms.

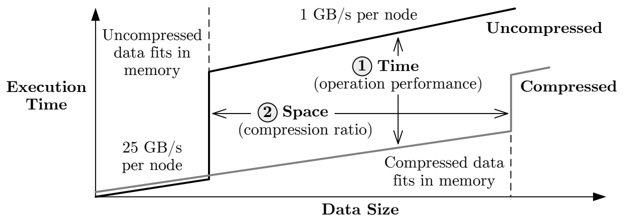


Figure 1: Goals of Compressed Linear Algebra.

Bandwidth Challenge: Many ML algorithms are iterative, with repeated read-only access to the data. They rely on matrix-vector multiplications to converge to an optimal model; such operations require one complete scan of the matrix, with two floating point operations per matrix element. Hence, matrix-vector multiplications are, even in-memory, I/O bound. Disk bandwidth is usually 10x-100x slower than memory bandwidth, so it it crucial for performance to fit the matrix into available memory without sacrificing operations performance. This challenge applies to single-node in-memory computations [16], data-parallel frameworks with distributed caching such as Spark [22], and hardware accelerators like GPUs, with limited device memory [1, 3, 4].

Compressed Linear Algebra: Declarative ML provides data independence, which allows for automatic lossless compression to fit larger datasets into memory. A baseline solution would employ general-purpose compression techniques and decompress matrices block-wise for each operation. However, heavyweight techniques like Gzip are inapplicable because decompression is too slow (slower than uncompressed I/O), while lightweight methods like Snappy only achieve moderate compression ratios. Existing special-purpose compressed matrix formats with good performance like CSR-VI [18] similarly show only modest compression ratios. We have therefore initiated the study of *compressed linear algebra (CLA)*, in which lightweight database compression methods—such as compressing offset lists per distinct column value—are applied to matrices and then linear algebra operations are executed directly on the compressed representations [12]. 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. The novelty of our approach is to combine both database compression techniques and sparse matrix representations, leading towards a generalization of traditional sparse matrix formats for sparse and dense data; see [12] for a full discussion of related work.

© VLDB Endowment 2016. This is a minor revision of the paper entitled Compressed Linear Algebra for Large-Scale Machine Learning, published in the Proceedings of the VLDB Endowment, Vol. 9, No. 12, 2150-8097/16/08. DOI: https://doi.org/10.14778/2994509.2994515.

Table 1: Compression Ratios of Real Datasets.

Dataset	Size	Gzip	Snappy	CLA
Higgs [19]	11M×28, 0.92: 2.5 GB	1.93	1.38	**2.03**
Census [19]	2.5M×68, 0.43: 1.3 GB	17.11	6.04	**27.46**
Covtype [19]	600K×54, 0.22: .14 GB	10.40	6.13	**12.73**
ImageNet [8]	1.2M×900, 0.31: 4.4 GB	5.54	3.35	**7.38**
Mnist8m [7]	8.1M×784, 0.25: 19 GB	4.12	2.60	**6.14**

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

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

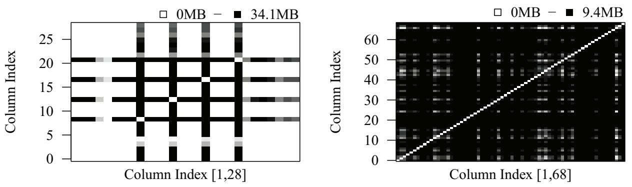
Compression Potential: Our focus is on floating-point matrices, so the potential for compression may not be obvious. Table 1 shows compression ratios for the general-purpose, heavyweight Gzip and lightweight Snappy algorithms and for our CLA method on real-world datasets (sizes given as rows, columns, 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 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. In the following sections, we motivate our approach and describe its key components: column compression schemes, cache-conscious vector-matrix operations, and an efficient sampling-based compression algorithm.

2. BACKGROUND AND MOTIVATION

As discussed below, both the features of declarative-ML systems and the characteristics of typical ML workloads motivate our approach to compressed linear algebra.

SystemML Setting: We describe CLA in the setting of SystemML, as it is representative of the declarative ML platforms that we are targeting. In SystemML, algorithms are expressed in a high-level R-like scripting language and compiled to hybrid runtime plans that combine both single-node, in-memory operations and distributed operations on MapReduce or Spark. Each statement block of an ML script is first parsed into a directed cyclic graph (DAG) of high-level operators. The system then applies various rewrites, such as common subexpression elimination and optimization of matrix-multiplication chains, as well as operator selection, yielding a DAG of low-level operators, which is then compiled into instructions. Matrices are represented internally in a binary *block matrix* format with fixed-size blocks. Each block is represented either in dense or sparse format. For single-node, in-memory operations, the entire matrix is often represented as a single block. CLA can be seamlessly integrated by adding a new derived block representation and operations. See [6, 12] for further details on SystemML.

Common Operation Characteristics: Table 2 summarizes the core operations of important ML algorithms.



(a) Co-Coding Higgs (b) Co-Coding Census

Figure 2: Cardinality Ratios and Co-Coding.

These algorithms include linear regression via iterative conjugate-gradient descent (LinregCG) and via direct solution of the normal equations (LinregDS), as well as logistic regression (Logreg), generalized linear models (GLM), support-vector machines with L_2-regularization (L2SVM) and principal component analysis (PCA). LinregDS and PCA are non-iterative and the other algorithms are iterative. Vector-matrix multiplication is often caused by the rewrite $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$ to avoid transposing $\mathbf{X}$. In addition, many systems also implement physical operators for matrix-vector chains, with optional element-wise weighting $\mathbf{w}\odot$, and transpose-self matrix multiplication (TSMM) $\mathbf{X}^\top\mathbf{X}$. 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 certain common characteristics of ML datasets. First, matrices usually have significantly more rows (observations) than columns (features), especially in enterprise machine learning, where data often originates from data warehouses. Second, feature preprocessing like dummy coding often yields datasets having many *sparse* features (i.e., features with many zero values); sparsity, however, is rarely uniform, but often varies widely among features [12]. Third, Many datasets contain features with low *column cardinality*, i.e., few distinct values. Examples include encoded categorical, binned or dummy-coded (0/1) features. Low column cardinality is a good indicator of compression potential because it indicates redundancy. For example, all columns of *Census* have a ratio of column cardinality to number of rows below .0008% and the majority of columns of *Higgs* have a cardinality ratio below 1%. The column cardinalities can vary widely within a dataset, however; for example, *Higgs* contains several columns having millions of distinct values. (See [12] for additional discussion of the datasets.) Finally, many datasets contain column groups that exhibit significant *correlation* in that the concatenated columns have a cardinality ratio much lower than would be expected if the values in each column were arranged randomly and independently of the other columns.

Compression Strategy: The foregoing workload characteristics suggest several key features of a good compression strategy. First, the compression schemes should be column-based and value-centric, with fallbacks for high cardinality columns. Moreover, schemes should exploit column correlation by discovering and *co-coding* highly correlated column groups. With value-based offset lists, a column i with d_i distinct values requires $\approx 8d_i + 4n$ 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

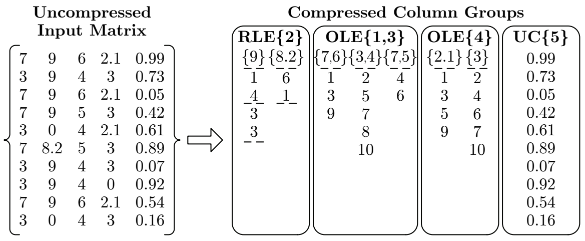


Figure 3: Example Compressed Matrix Block.

group of value-pairs and offsets requires $16d_{ij} + 4n$ B, where d_{ij} is the number of distinct value-pairs. The higher the correlation, the larger the size reduction by co-coding. For example, Figures 2(a) and 2(b) show the size reductions (in MB) by co-coding all pairs of columns of *Higgs* and *Census*. Overall, co-coding column groups of *Census* (not limited to pairs) improves the compression ratio from 10.1x to 27.4x. 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.

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

A compressed matrix block is represented 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. As with sparse matrix formats, offset lists allow for efficient linear algebra operations.

Column Co-Coding: We exploit column correlation 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). We fall back to a simple uncompressed-column (UC) format 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—i.e., automatically choosing plans that maximize the compression ratio—in Section 4.

Example Compressed Matrix: Figure 3 shows our running example of a compressed matrix block. The 10×5 input matrix is represented as four column groups. 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.

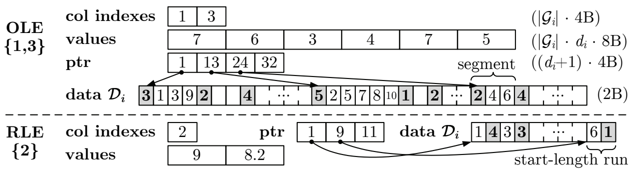


Figure 4: Data Layout OLE/RLE Column Groups.

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 $\mathbf{t}_{ij}$ $(1 \le j \le d_i)$. We focus on the "sparse" case in which zero values are not stored (aka "0-suppressing"). Also, denote by α the size in bytes of each floating point value; $\alpha = 8$ for the double-precision IEEE-754 standard.

3.2 Column Encoding Formats

We now describe the compressed data layout of the OLE and RLE 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 4 shows—as an extension to our running example from Figure 3 (with more rows)—the data layout of OLE/RLE column groups composed of four linearized arrays. Besides a data array $\mathcal{D}_i$, both encoding schemes use a common header of three arrays for column indexes, fixed-length value tuples, and pointers to the data per tuple. 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 $13 - 1 = 12$). 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.

Offset-List Encoding (OLE): Our OLE scheme divides the offset range into *segments* of fixed length $\Delta^{\mathrm{s}} = 2^{16}$ (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 $(= 1 + \lfloor (155{,}762 - 1)/\Delta^{\mathrm{s}} \rfloor)$ and is encoded as 24,690 $(= 155{,}762 - 2\Delta^{\mathrm{s}})$. 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

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

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

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 $\mathbf{q}$, $\pi_{ij}$
```

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 as

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

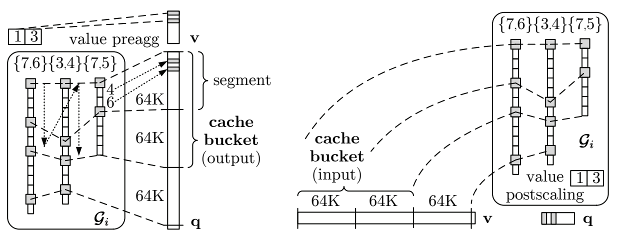
where r_{ij} is the number of runs for tuple $\mathbf{t}_{ij}$. Again, the common header has a size of $4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big)$.

3.3 Operations over Compressed Matrices

We now show how to execute efficient linear algebra operations over a set $\mathcal{X}$ of column groups; matrix block operations are then composed of operations over column groups. We write $c\mathbf{v}$ to denote element-wise scalar-vector multiplication as well as $\mathbf{u} \cdot \mathbf{v}$ to denote the inner product of vectors.

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 projection of $\mathbf{v}$ onto 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 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 5(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 $\Delta^{\text{c}} \cdot \#\text{cores} \cdot 8\,\text{B}$ fits in L3 cache, by default $\Delta^{\text{c}} = 2\Delta^{\text{s}}$), 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.

As a toy example for OLE, consider the column group $\mathcal{G} = \{1, 3\}$ as in Figure 4 and suppose that $\mathbf{v}_{\mathcal{G}} = (1, 2)$. Also suppose that the OLE encoding uses two segments, each of length = 5 rows, and that a cache bucket comprises exactly one segment. Finally, suppose that a single thread



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

Figure 5: Cache-Conscious OLE Operations.

updates $\mathbf{q}$. Algorithm 1 first precomputes $(1, 2) \cdot (7, 6) = 19$, $(1, 2) \cdot (3, 4) = 11$, and $(1, 2) \cdot (7, 5) = 17$. The thread then handles rows in $[rl, ru) = [1, 11)$, i.e., all ten rows. It reads the first five elements of $\mathbf{q}$ into cache, and then adds 19 to $\mathbf{q}[1]$ and $\mathbf{q}[3]$, 11 to $\mathbf{q}[2]$ and $\mathbf{q}[5]$, and 17 to $\mathbf{q}[4]$. Next, the thread reads in the last five elements of $\mathbf{q}$ and adds 19 to $\mathbf{q}[9]$, 11 to $\mathbf{q}[7]$, $\mathbf{q}[8]$, $\mathbf{q}[10]$, and 17 to $\mathbf{q}[6]$. In contrast, the naïve approach would first add 19 to $\mathbf{q}[i]$ for $i = 1, 3, 9$, then add 11 to $\mathbf{q}[i]$ for $i = 2, 5, 7, 8, 10$, and then add 17 to $\mathbf{q}[i]$ for $i = 4, 6$. The cost on our toy architecture is six "cache reads" compared to two reads for Algorithm 1. Also note that Algorithm 1 requires only 6 multiplications and 13 additions, whereas the uncompressed operation requires 20 multiplications and 20 additions.

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 5(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 $2\Delta^{\text{s}}$) except that RLE runs are allowed to cross cache bucket boundaries due to column-wise parallelization.

Other Operations: As discussed in [12], efficient methods for more complex operations such as matrix-vector multiplication chains and transpose-self matrix multiplications are built up from the foregoing matrix-vector and vector-matrix operations. Common operations such as $\mathbf{X}^2$, $2\mathbf{X}$, and `append` can be executed very efficiently over compressed matrices without scanning the offset lists. Finally, unary aggregates like `sum` (or similarly `colSums`) are efficiently computed using offset-list sizes as $\sum_{i=1}^{|\mathcal{X}|}\sum_{j=1}^{d_i}|\mathcal{O}_{ij}|\mathbf{t}_{ij}$.

4. COMPRESSION PLANNING

Given an uncompressed $n \times m$ matrix block $\mathbf{X}$, the system automatically chooses 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, sampling-based techniques are used to estimate 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. Exhaustive ($O(m^m)$) and brute-force greedy ($O(m^3)$) partitioning are infeasible, but a bin-packing-based technique can drastically reduce the number of candidate groups. The overall compression algorithm corrects for estimation errors.

4.1 Estimating Compressed Size

To calculate the compressed size of a column group $\mathcal{G}_i$ via size-estimation formulas (1) and (2), 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 $\mathbf{X}$ with $|\mathcal{S}| \ll n$. We have found experimentally that being conservative (overestimating compressed size) and correcting later on yields the most robust co-coding choices, so we make conservative choices in our estimator design.

Number of Distinct Tuples: To estimate d_i, we use the "hybrid" estimator $\hat{d}_i$ from [14]; 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 and then apply a "generalized jackknife" estimator that performs well for that regime. 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"). The hybrid estimator provides a reasonable balance of cost and accuracy [12].

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 [13]), 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 un-

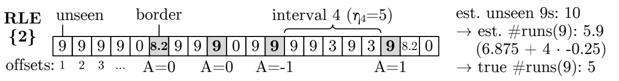


Figure 6: Estimating the Number of RLE Runs $\hat{r}_{ij}$.

observed 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 ($N_{ijk} \le l_k$). 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.

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 6. 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: $\hat{f}_{ijk}^u = (\eta_k/\eta)\hat{f}_{ij}^u$. 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 an "Ising-Stevens" distribution [17, pp. 422-423] and we estimate r_{ijk} by its mean: $\hat{r}_{ijk} = \hat{f}_{ijk}^u(\eta_k - \hat{f}_{ijk}^u + 1)/\eta_k$. We show in [12] that a reasonable estimate of the contribution to r_{ij} from the border between interior intervals k and $k+1$ is $\hat{A}_{ijk} = 1 - (2\hat{f}_{ij}^u/\eta)$, so that the final estimate is $\hat{r}_{ij} = \sum_k \hat{r}_{ijk} + \sum_k \hat{A}_{ijk}$ (with appropriate modifications for the first and last border).

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. Al-

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($C^{\mathrm{UC}}$)
```

though 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 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 the weight of the i^{th} column to $\hat{d}_i/n$, i.e., the ratio of distinct tuples to rows. If $\hat{d}_i/n$ is larger than a specified threshold γ, then we consider column i as ineligible for grouping. We also set each bin capacity to $w = \beta\gamma$, where β is a tuning parameter. 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 for common scenarios, 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 $\hat{S}_i^{\mathrm{UC}}/\hat{S}_i^{\mathrm{C}}$ 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 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 present some highlights from an experimental study of CLA as implemented in SystemML, emphasizing end-to-end results; see [12] for details and additional experiments. Overall, the results show that, for a variety of ML programs and real-world datasets, CLA indeed achieves in-memory matrix-vector multiplication performance close to uncompressed while yielding substantially better compression ratios than lightweight general-purpose compression. As a consequence, CLA provides large end-to-end performance improvements when uncompressed or lightweight-compressed matrices do not fit in local or aggregated memory.

Implementation: When CLA is enabled, SystemML automatically injects—for any multi-column input matrix—a so-called `compress` operator via new rewrites. This applies to both single-node and distributed Spark operations, where the execution type is chosen based on memory estimates. For Spark, 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.

Experimental Setup: We ran all experiments on a cluster of one head node and six additional nodes; see [12] for details. For our end-to-end experiments, we ran versions of