The VLDB Journal manuscript No.
(will be inserted by the editor)

Compressed Linear Algebra for Large-Scale Machine Learning

Ahmed Elgohary · Matthias Boehm · Peter J. Haas · Frederick R. Reiss · Berthold Reinwald

Received: date / Accepted: date

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 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. Therefore, we initiate work—inspired by database compression and sparse matrix formats—on value-based compressed linear algebra (CLA), in which heterogeneous, lightweight database compression techniques are applied to matrices, and then linear algebra operations such as matrix-vector multiplication are executed directly on the compressed representation. 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, which enables fitting substantially larger datasets into available memory. We thereby obtain significant end-to-end performance improvements up to 9.2x.

1 Introduction

Data has become a ubiquitous resource [24]. Large-scale machine learning (ML) leverages these large data collections in order to find interesting patterns and build robust predictive models [24,28]. Applications range from traditional regression analysis and customer classification to recommendations. In this context, data-parallel frameworks such as MapReduce [29], Spark [95], or Flink [4] often are used for cost-effective parallelized model training on commodity hardware.

Declarative ML: State-of-the-art, large-scale ML systems support declarative ML algorithms [17], expressed in high-level languages, that comprise linear algebra operations such as matrix multiplications, aggregations, element-wise and statistical operations. Examples—at different levels of abstraction—are SystemML [16], Mahout Samsara [78], Cumulon [41], DMac [93], Gilbert [75], SciDB [82], SimSQL [60], and TensorFlow [2]. A high level of abstraction gives data scientists the flexibility to create and customize ML algorithms without worrying about data and cluster characteristics, data representations (e.g., sparse/dense formats and blocking), or execution-plan generation.

Bandwidth Challenge: 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; such operations require one complete scan of the matrix, with two floating point operations per matrix element. Disk bandwidth is usually 10x-100x slower than memory bandwidth, which is in turn 10x-40x slower than peak floating point performance; so matrix-vector multiplication is, even in-memory, I/O bound. Hence, it is crucial for performance to fit the matrix into available memory without sacrificing operations performance. This challenge applies to single-node in-memory computations [42], data-parallel frameworks with distributed caching such as Spark [95], and hardware accelerators like GPUs, with limited device memory [2,7,11].

Matthias Boehm, Peter J. Haas, Frederick R. Reiss · Berthold Reinwald
IBM Research – Almaden; San Jose, CA, USA

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

Table 1 Compression Ratios of Real Datasets (which are used throughout this paper).

Dataset	**Size** ($n \times m$, sparsity nnz/$(n \cdot m)$, size)	**Gzip**	**Snappy**	**CLA** [32]	**CLA** (this paper)
Higgs [59]	11,000,000 × 28, 0.92, 2.5 GB	1.93	1.38	**2.03**	**2.17**
Census [59]	2,458,285 × 68, 0.43, 1.3 GB	17.11	6.04	**27.46**	**35.69**
Covtype [59]	581,012 × 54, 0.22, 0.14 GB	10.40	6.13	**12.73**	**18.19**
ImageNet [23]	1,262,102 × 900, 0.31, 4.4 GB	5.54	3.35	**7.38**	**7.34**
Mnist8m [19]	8,100,000 × 784, 0.25, 19 GB	4.12	2.60	**6.14**	**7.32**
Airline78 [5]	14,462,943 × 29, 0.73, 3.3 GB	7.07	4.28	N/A	**7.44**

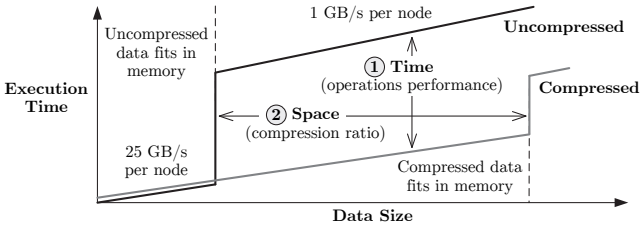


Fig. 1 Goals of Compressed Linear Algebra.

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 employ general-purpose compression techniques and decompress matrices blockwise for each operation. However, heavyweight techniques like Gzip are not applicable because decompression is too slow, while lightweight methods like Snappy achieve only moderate compression ratios. Existing special-purpose compressed matrix formats with good performance like CSR-VI [52] similarly show only modest compression ratios. In contrast, our approach builds upon research on lightweight database compression, such as compressed bitmaps and dictionary coding, as well as sparse matrix representations. Specifically, we initiate the study of value-based *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.

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 are given as rows, columns, sparsity—i.e., ratio of #non-zeros (nnz) to cells—and in-memory size. We observe compression ratios of 2.2x–35.7x, due to a mix of floating point and integer data, and due to features with relatively few distinct values. In comparison, previously published CLA results [32] showed compression ratios of 2x–27.5x and did not include the Airline78 dataset (years 2007 and 2008 of the Airline dataset [5]). Thus, unlike in scientific computing [14], 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 altogether.

Contributions: Our major contribution is to make a case for value-based *compressed linear algebra (CLA)*, 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 to combine both, leading toward a generalization of sparse matrix representations and operations. In this paper, we describe an extended version of CLA that improves previously published results [32]. 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 as a representative system, 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 provide an efficient sampling-based algorithm for selecting a good compression plan, including techniques for compressed-size estimation and column grouping.
- *Experiments:* In Section 5, we study—with CLA integrated into Apache SystemML—a variety of ML algorithms and real-world datasets in both single-node and distributed settings. We also compare CLA against alternative compression schemes.

– *Discussion and Related Work:* Finally, we discuss limitations and open research problems, as well as related work, in Sections 6 and 7.

Extensions of Original Version: Apart from providing a more detailed discussion, in this paper we extend the original CLA framework [32] in several ways, improving both compression ratios and operations performance. The major extensions are (1) the additional column encoding format DDC for dense dictionary coding, (2) a new greedy column grouping algorithm with pruning and memoization, (3) additional operations including row aggregates and multi-threaded compression, and (4) hardened size estimators. Furthermore, we transferred the original CLA framework into Apache SystemML 0.11 and the extensions of this paper into SystemML 0.14. For the sake of reproducible results, we accordingly repeated all experiments on SystemML 0.14 using Spark 2.1, including additional datasets, micro benchmarks, and end-to-end experiments.

2 Background and Motivation

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

2.1 SystemML Architecture

SystemML [16,34] aims at declarative ML [17], where algorithms are expressed in a high level language with R-like syntax and compiled to hybrid runtime plans [42] that combine single-node, in-memory operations and distributed operations on MapReduce or Spark. We outline the features of SystemML relevant to CLA.

ML Program Compilation: An ML script is first parsed into a hierarchy of statement blocks that 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 simplifications, 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 that 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 [41], chunks [82], or blocks [16,60,75,94], are widely used in existing large-scale ML systems. Each block may be represented in either dense or sparse format to allow for block-local decisions and efficiency on datasets with non-uniform sparsity. SystemML uses a modified CSR (compressed sparse row), CSR, or COO (coordinate) format for sparse or ultra-sparse blocks. For single-node, in-memory operations, the entire matrix is represented as a single block [42] to reuse data structures and 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 workload characteristics of linear algebra operations and matrix properties.

An Example: Consider the task of fitting a simple linear regression model via the conjugate gradient (CG) method [34,63]. The LinregCG algorithm reads a feature matrix $\mathbf{X}$ and a continuous label vector $\mathbf{y}$, 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:

```
1:  X = read($1);               # n x m feature matrix
2:  y = read($2);               # n x 1 label vector
3:  maxi = 50; lambda = 0.001;  # t(X)..transpose of X
4:  r = -(t(X) %*% y); ...      # %*%..matrix multiply
5:  norm_r2 = sum(r * r); p = -r;   # initial gradient
6:  w = matrix(0, ncol(X), 1); i = 0;
7:  while(i < maxi & norm_r2 > norm_r2_trgt) {
8:     # compute conjugate gradient
9:     q = ((t(X) %*% (X %*% p)) + lambda * p);
10:    # compute step size
11:    alpha = norm_r2 / sum(p * q);
12:    # update model and residuals
13:    w = w + alpha * p;
14:    r = r + alpha * q;
15:    old_norm_r2 = norm_r2;
16:    norm_r2 = sum(r^2); i = i + 1;
17:    p = -r + norm_r2/old_norm_r2 * p; }
18: 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 costs). This is especially true with hybrid runtime plans (see Section 2.1), where

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

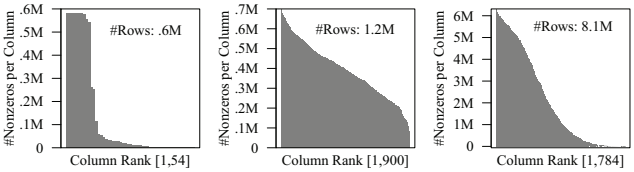
Algorithm	M-V $\mathbf{Xv}$	V-M $\mathbf{v}^\top\mathbf{X}$	MVChain $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{Xv}))$	TSMM $\mathbf{X}^\top\mathbf{X}$
LinregCG	✓	✓	✓ (w/o $\mathbf{w}\odot$)	
LinregDS		✓		✓
Logreg	✓	✓	✓ (w/ $\mathbf{w}\odot$)	
GLM	✓	✓	✓ (w/ $\mathbf{w}\odot$)	
L2SVM	✓	✓		
PCA	✓			✓

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 computing $\mathbf{X}^\top$ is expensive, whereas computing $\mathbf{v}^\top$ involves only a metadata update. Many systems also implement physical operators for matrix-vector chains (e.g., line 9) with optional element-wise weighting $\mathbf{w}\odot$, and transpose-self matrix multiplication (TSMM) $\mathbf{X}^\top\mathbf{X}$ [7,42,78]. All of these operations are I/O-bound, except for TSMM with $m \gg 1$ features because its compute workload grows as $O(m^2)$. Other common operations over $\mathbf{X}$ are `append`, unary aggregates like `colSums`, and matrix-scalar operations for intercept computation, scaling, and shifting.

Common Data Characteristics: Despite significant differences in data sizes—ranging from kilo- to terabytes—input data for the aforementioned algorithm classes share common data characteristics:

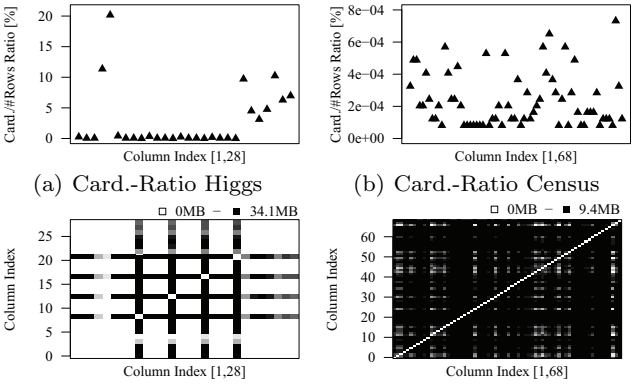
- *Tall and Skinny Matrices:* Matrices usually have significantly more rows (observations) than columns (features), especially in enterprise ML [7,96], where data often originates from data warehouses.
- *Non-Uniform Sparsity:* Sparse datasets usually have many features, often created via pre-processing, e.g., dummy coding.[1] Sparsity, however, is rarely uniform, but varies among features. For example, Figure 2 shows the sparsity skew of our sparse datasets.
- *Low Column Cardinalities:* Many datasets exhibit features with few distinct values, e.g., encoded categorical, binned or dummy-coded features.
- *Column Correlations:* Correlation among features is also very common and typically originates from natural data correlation, use of composite features such as interaction terms in a regression model (e.g., the second term in $y = ax_1 + bx_1x_2$), or again preprocessing techniques like dummy coding.

[1] Dummy coding transforms a categorical feature having d possible values into d boolean features, each indicating the rows in which a given value occurs. The larger the value of d, the greater the sparsity (from adding $d-1$ zeros per row).



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

Fig. 2 Sparsity Skew (Non-Uniform Sparsity).



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

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

Fig. 3 Cardinality Ratios and Co-Coding.

The foregoing four 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, independent of value representations. 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 ratio below .0008% and the majority of columns of Higgs have a 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. For value-based offset lists, a column i with d_i distinct values—including zeros—requires $\approx 8d_i + 4n$ bytes, where n is the number of rows, and each value is encoded with 8 bytes plus a list of 4-byte row indexes,

which represent commonly used types of blocked matrices [16,94]. Co-coding two columns i and j as a single group of value-pairs and offsets requires $16d_{ij} + 4n$ bytes, where d_{ij} is the number of distinct value-pairs. The larger the correlation $\max(d_i, d_j)/d_{ij}$, 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) improved the compression ratio from 12.8x to 35.7x. We therefore try to discover and co-code column groups.

2.4 Lightweight DB Compression and Sparse Formats

To facilitate understanding of our matrix compression schemes, we briefly review common database compression techniques and sparse matrix formats.

Lightweight Database Compression: Modern column stores typically apply lightweight database compression techniques [1,56,72,90,97]. For a recent experimental analysis, see [27]. Common schemes include (1) dictionary encoding, (2) null suppression, (3) run-length encoding (RLE), (4) frame-of-reference (FOR), (5) patched FOR (PFOR), as well as (6) bitmap indexes and compression. The prevalent schemes, however, are dictionary and run-length encoding. Dictionary encoding creates a dictionary of distinct values and replaces each data value with a (smaller) dictionary reference. In contrast, RLE represents runs of consecutive entries with equal value as tuples of value, run length, and, optionally, starting position.

Sparse Matrix Formats: Probably the most widely used sparse formats are CSR and CSC (compressed sparse rows/columns) [44,64,76], which encode non-zero values as index-value pairs in row- and column-major order, respectively. For example, the basic CSR format uses an $(n + 1)$-length array for row pointers and two $O(\text{nnz})$ arrays for column indexes and values; a row pointer comprises the starting positions of the row in the index and value arrays, and column indexes per row are ordered for binary search.

3 Compression Schemes

We now describe our novel matrix compression framework, including several effective encoding formats for compressed column groups, as well as efficient, cache-conscious 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, or *dictionary*, together with either a list of *offsets* per value—i.e., a list of row indexes in which the value appears—or a list of *references* to distinct values, where a reference to a value gives the value's position in the dictionary. We shall show that, similar to sparse and dense matrix formats, these formats allow for efficient linear algebra operations.

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 that represent reals or integers.

Column Encoding Formats: Each offset list and each list of tuple references is stored in a compressed representation, and the efficiency of executing linear algebra operations over compressed matrices strongly depends on how fast we can iterate over this representation. We adapt several well-known effective offset-list and dictionary encoding formats:

- *Offset-List Encoding* (OLE) is inspired by sparse matrix formats and encodes the offset lists per value tuple as an ordered list of row indexes.
- *Run-Length Encoding* (RLE) is inspired by sparse bitmap compression and encodes the offset lists per value tuple as sequence of runs representing starting row indexes and run lengths.
- *Dense Dictionary Coding* (DDC) is inspired by dictionary encoding and stores tuple references to the list of value tuples including zeros.
- *Uncompressed Columns* (UC) is used as a fallback if compression is not beneficial; the set of uncompressed columns is stored as a sparse or dense block.

Different column groups may be compressed using different encoding formats. The best co-coding and formatting choices are strongly data-dependent and hence require automatic optimization. We discuss sampling-based compression planning in Section 4.

Example Compressed Matrix: Figure 4 shows our running example of a compressed matrix block in its logical representation. The 10×5 input matrix is encoded as four column groups, where we use 1-based indexing for row indexes and tuple references. Columns

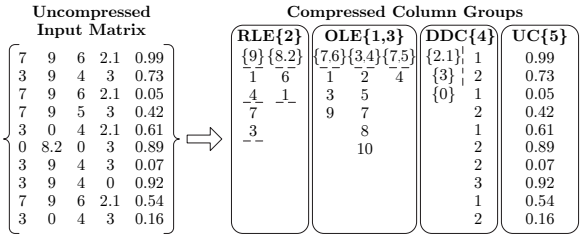


Fig. 4 Example Compressed Matrix Block.

2, 4, and 5 are represented as single-column groups and encoded via RLE, DDC, and UC, respectively. For column 2 in RLE, we have two distinct non-zero values and hence two associated offset lists encoded as runs, which represent starting-row indexes and run lengths. Column 4 in DDC has three distinct values (including zero) and encodes the data as tuple references, whereas column 5 is a UC group in dense format. Finally, there is a co-coded OLE column group for the correlated columns 1 and 3, which encodes offset lists for all three distinct non-zero value-pairs as lists of row indexes.

Notation: For the ith column group, denote by $\mathcal{T}_i = \{\mathbf{t}_{i1}, \mathbf{t}_{i2}, \ldots, \mathbf{t}_{id_i}\}$ the set of d_i distinct tuples, by $\mathcal{G}_i$ the set of column indexes, and by $\mathcal{O}_{ij}$ the set of offsets associated with $\mathbf{t}_{ij}$ $(1 \leq j \leq d_i)$. The OLE and RLE schemes are "sparse" formats in which zero values are not stored (0-suppressing), whereas DDC is a dense format, which includes zero values. Also, denote by α the size in bytes of each floating point value, where $\alpha = 8$ for the double-precision IEEE-754 standard.

3.2 Column Encoding Formats

Figure 5 provides an overview of the OLE, RLE, DDC, and UC representations used in our framework. The UC format stores a set of columns as an uncompressed dense or sparse matrix block. In contrast, all of our compressed formats are value-based, i.e., they store a dictionary of distinct tuples and a mapping between tuples and rows in which they occur. OLE and RLE use *offset lists* to map from value tuples to row indexes. This is especially effective for sparse data, but also for dense data with runs of equal values. DDC uses *tuple references* to map from row indexes to value tuples, and is effective for dense data with few distinct items and

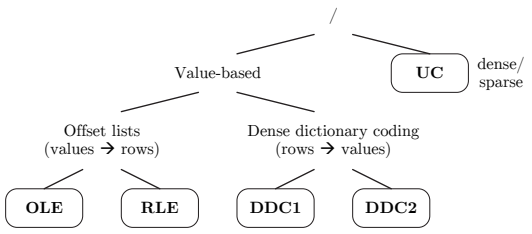


Fig. 5 Overview of Column Encoding Formats.

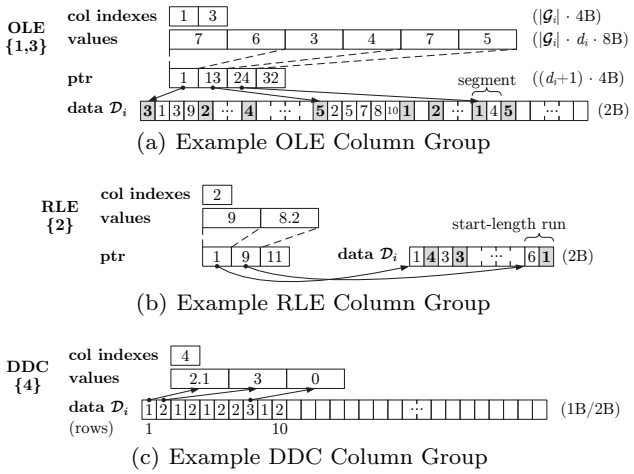


Fig. 6 Data Layout of Compressed Column Groups.

few runs. We use two versions, DDC1 and DDC2, with 1 and 2 byte references for dictionaries with $d_i \leq 255$ and $d_i \leq 65{,}535$ non-zero tuples, respectively. We now describe the physical data layout of these encoding formats and give formulas for the in-memory compressed size S_i^{OLE}, S_i^{RLE}, and S_i^{DDC}. The matrix size is then computed as the sum of column group size estimates.

Data Layout: Figure 6 shows—as an extension to our running example from Figure 4 (with more rows)—the data layouts of OLE, RLE, and DDC column groups, each composed of up to four arrays. All encoding formats use a common header of two arrays for column indexes and fixed-length value tuples, as well as a data array $\mathcal{D}_i$. Tuples are stored in order of decreasing value frequency to improve branch prediction, cache locality, and pre-fetching. The header of OLE and RLE groups further contains an array for 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) because the encoded offset lists are stored consecutively. The data array is then used in an encoding-specific manner.

Offset-List Encoding (OLE): Our OLE format divides the offset range into *segments* of fixed length $\Delta^{\text{s}} = 2^{16}$ in order to encode each offset with only two bytes. 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 $\mathcal{D}_i$. For example, in Figure 6(a), the nine instances of $\{7, 6\}$ appear in three consecutive segments, which gives a total length of 12. 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

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

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

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, as well as reconstructing and enumerating global offsets per run. The size S_i^{RLE} of column group $\mathcal{G}_i$ is calculated as

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

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

Dense Dictionary Coding (DDC): The DDC format uses a dense, fixed-length data array $\mathcal{D}_i$ of n entries. An entry at position k represents the kth row as a reference to tuple $\mathbf{t}_{ij}$, encoded as its position in the dictionary, which includes zero if present. Therefore, the size of the dictionary—in terms of the number of distinct tuples d_i—determines the physical size of each entry. In detail, we use two byte-aligned formats, DDC1 and DDC2, with one and two bytes per entry. Accordingly, these DDC formats are only applicable if $d_i \leq 2^8 - 1$ or $d_i \leq 2^{16} - 1$. The total size S_i^{DDC} of column group $\mathcal{G}_i$ is calculated as

$$S_i^{\text{DDC}} = \begin{cases} 4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i| + n & \text{if } d_i \leq 2^8 - 1 \\ 4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i| + 2n & \text{if } 2^8 \leq d_i \leq 2^{16} - 1, \end{cases} \quad (3)$$

where $4|\mathcal{G}_i| + d_i\alpha|\mathcal{G}_i|$ denotes the header size of column indexes and the dictionary of value tuples.

Overall, these column encoding formats encompass a wide variety of dense and sparse data as well as special data characteristics. Because they are all value-based formats, column co-coding and common runtime techniques apply similarly to all of them.

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 composed of operations over column groups, facilitating simplicity and extensibility with regard to compression plans of heterogeneous 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.

Overview of Techniques: Table 3 provides an overview of applied, encoding-format-specific techniques. This includes pre-aggregation, post-scaling, and cache-conscious techniques (*):

- *Pre-Aggregation:* The matrix-vector multiplication $\mathbf{q} = \mathbf{X}\mathbf{v}$ 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, the value-based representation of compressed column groups allows pre-computation of $\mathbf{u}_{ij} = \mathbf{t}_{ij}\cdot\mathbf{v}_{\mathcal{G}_i}$ once for each tuple $\mathbf{t}_{ij}$. The more columns co-coded and the fewer distinct tuples, the more this pre-aggregation reduces the number of required floating point operations.
- *Post-Scaling:* The vector-matrix product $\mathbf{q} = \mathbf{v}^\top\mathbf{X}$ can be written as $\mathbf{q}_{\mathcal{G}_i} = \sum_{j=1}^{d_i}\sum_{l\in\mathcal{O}_{ij}} v_l\mathbf{t}_{ij}$ for $\mathcal{G}_i \in \mathcal{X}$. We compute this as $\mathbf{q}_{\mathcal{G}_i} = \sum_{j=1}^{d_i}\mathbf{t}_{ij}(\mathbf{v}\cdot\mathbf{1}_{\mathcal{O}_{ij}})$, i.e., we sum up input-vector values according to the offset list per tuple, scale this sum only once with each value in $\mathbf{t}_{ij}$ (per column), and add the results to the corresponding output entries.

Both pre-aggregation and post-scaling are distributive law rewrites of sum-product optimization [31]. Since multi-threaded vector-matrix multiplication is parallelized over column groups, post-scaling also avoids *false sharing* [18] if multiple threads would update disjoint entries of the same cache lines. Furthermore, UC column groups are separately parallelized to avoid load imbalance in case of large uncompressed groups.

Table 3 Overview of Basic Techniques.

Format	Matrix-Vector	Vector-Matrix
OLE	Pre-aggregation, horiz. OLE scan*	Post-scaling, horiz. OLE scan*
RLE	Pre-aggregation, horiz. RLE scan*	Post-scaling, horiz. RLE scan*
DDC	Pre-aggregation, DDC1 blocking*	Post-scaling
Multi-threading	row partitions, UC separate	column groups, UC separate

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** Δ^c **do** *// cache partitions in* $[rl, ru)$
5: **for** j **in** $[1, d_i]$ **do** *// distinct tuples*
6: **for** k **in** $[bk, \min(bk + \Delta^c, ru))$ **by** Δ^s **do** *// segm.*
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* $\mathbf{q}$, π_{ij}

Matrix-Vector Multiplication: Despite pre-aggregation, 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*; see Algorithm 1 and Figure 7(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 partition of size Δ^c (such that $\Delta^c \cdot \alpha \cdot \#\text{cores}$ 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. In contrast, DDC does not require horizontal scans but allows—due to random access—cache blocking across multiple DDC groups. However, we apply cache blocking only for DDC1 because its temporary memory requirement is bound by 2 KB per group.

Vector-Matrix Multiplication: Similarly, despite post-scaling, pure column-wise processing would suffer from cache-unfriendly behavior because we would scan the input vector $\mathbf{v}$ once for each distinct tuple. Our cache-conscious OLE/RLE group operations therefore again use *horizontal, segment-aligned scans* as shown in Figure 7(b). Here we sequentially operate on cache partitions of $\mathbf{v}$. The OLE, RLE, and DDC algorithms are similar to matrix-vector multiplication, but in the inner loop we sum up input-vector values according to the given offset list or references, and finally, scale the aggregates once with the values in $\mathbf{t}_{ij}$. For multi-threaded operations, we parallelize over column groups, where disjoint results per column allow for simple dynamic task scheduling. The cache-partition size for OLE and RLE is equivalent to matrix-vector (by default $2\Delta^s$) except that RLE runs are allowed to cross partition boundaries due to column-wise parallelization.

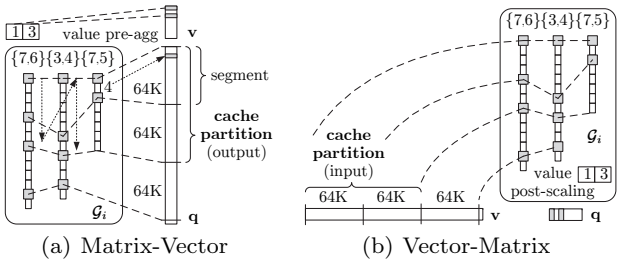


Fig. 7 Cache-Conscious OLE Operations.

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 $\mathbf{u} = \mathbf{w} \odot \mathbf{q}$, 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 $\mathbf{p} = \mathbf{v}_k^\top \mathcal{X}_{i \leq j}$, where the condition $i \leq j$ exploits the symmetry of $\mathbf{X}^\top\mathbf{X}$. Vectors originating from single-column DDC groups are not decompressed because DDC allows random access and hence efficient vector-matrix multiplication. Each non-zero output cell $\mathbf{p}_l$ is written to the upper triangular matrix only, i.e., to $\mathbf{R}_{k,l}$ if $k \leq l$ and $\mathbf{R}_{l,k}$ otherwise. Finally, we copy the upper triangle in a cache-conscious manner to the lower triangle. Multi-threaded operations parallelize over ranges of column groups.

Other Operations: Various common operations can be executed very efficiently over compressed matrices without scanning the entire data. In general, this includes value-based operations that leave the non-zero structure unchanged and operations that are efficiently computed via counts. Example operations include:

- *Matrix-Scalar Operations:* Sparse-safe matrix-scalar operations—i.e., operations that can safely ignore zero inputs—such as element-wise matrix power $\mathbf{X}^c$ or element-wise multiplication $c\mathbf{X}$ are carried out—for all encoding formats—with a single pass over the set of tuples $\mathcal{T}_i$ for each column group $\mathcal{G}_i$. For sparse-unsafe operations, DDC groups similarly process only the value tuples because zeros are represented, whereas OLE and RLE compute the new value for zeros once, determine a zero indicator vector, and finally create a modified compressed group.

– *Unary Aggregates:* We compute aggregates like `sum` or `colSums` 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, whereas for DDC, we count occurrences per tuple reference. These counts are then scaled by the values to compute the aggregate. In contrast, `min`, `max`, `colMins`, and `colMaxs` are computed—given zero indicators per column—over value tuples without accessing the data. Row aggregates such as `rowSums`, `rowMins`, or `rowMaxs` are again computed in a cache-conscious manner.
– *Statistical Estimates:* Similar to unary aggregates, unweighted order statistics such as `quantile` or `median`, and higher-order statistics such as `moment` can be efficiently computed via counts. The idea is to use the distinct value tuples—padded with a single zero entry if necessary—and their corresponding counts as input to existing uncompressed weighted statistics, where we use the counts as weights.
– *Append Operations:* Finally, `cbind` operations that append another matrix column-wise to $\mathbf{X}$—as commonly used for intercept computation, where we append a column of 1's—are done via simple concatenation of column groups by reference.

Although the operation efficiency depends on the individual encoding formats, we see common characteristics because all of our formats are value-based.

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, or DDC 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 two techniques for column partitioning, including a new bin-packing-based technique, as well as an efficient greedy algorithm with pruning and memoization. Together, these techniques drastically reduce 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, nonzero 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), (2) and (3). The 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: Sampling-based estimation of the number of distinct tuples d_i is a well studied but challenging problem [21,37,73,87]. We have found that the *hybrid* estimator [37] is satisfactory for our purposes, compared to more expensive estimators like KMV [12] or Valiants' estimator [87]. The idea is to first estimate the degree of variability in the population frequencies of the tuples in $\mathcal{T}_i$ as low, medium, or high, as measured by the squared coefficient of variation $\gamma^2 = (1/d)\sum_{j=1}^{d}(n_j-\overline{n})^2/\overline{n}^2$, where n_j is the population frequency of the jth tuple in $\mathcal{T}_i$ and $\overline{n} = (1/d)\sum_{j=1}^{d} n_j = n/d$ is the average tuple frequency. It is shown in [37] that γ^2 can be estimated by $\hat{\gamma}^2(\hat{d}) = \max(\hat{\gamma}_0^2(\hat{d}), 0)$, where $\hat{\gamma}_0^2(\hat{d}) = (\hat{d}/|\mathcal{S}|^2)\sum_{j=1}^{|\mathcal{S}|} j(j-1)h_j + (\hat{d}/n) + 1$; here $\hat{d}$ is an estimator of d and h_j is the number of tuples that appear exactly j times in the sample ($1 \le j \le |\mathcal{S}|$). We then apply a "generalized jackknife" estimator that performs well for the respective variability regime to obtain an estimate $\hat{d}_i$ with [37]:

$$\hat{d}_{\text{hybrid}} = \begin{cases} \hat{d}_{\text{uj2}} & \text{if } 0 \le \hat{\gamma}^2(\hat{d}_{\text{uj1}}) < \alpha_1 \\ \hat{d}_{\text{uj2a}} & \text{if } \alpha_1 \le \hat{\gamma}^2(\hat{d}_{\text{uj1}}) < \alpha_2 \\ \hat{d}_{\text{Sh3}} & \text{if } \alpha_2 \le \hat{\gamma}^2(\hat{d}_{\text{uj1}}), \end{cases} \tag{4}$$

where $\hat{d}_{\text{uj1}}$, $\hat{d}_{\text{uj2}}$, $\hat{d}_{\text{uj2a}}$, and $\hat{d}_{\text{Sh3}}$ denote the unsmoothed first- and second-order jackknife estimators, the stabilized unsmoothed second-order jackknife estimator, and a modified Shlosser estimator, respectively. These estimators have the general form

$$\hat{d} = d_{\mathcal{S}} + K(h_1/|\mathcal{S}|), \tag{5}$$

where $d_{\mathcal{S}}$ is the number of distinct tuples in the sample and K is a constant computed from the sample. For example, the basic $\hat{d}_{\text{uj1}}$ estimator is derived using a first-order approximation under which each n_j is assumed to equal n/d, leading to a value of $K = d(1-q)$, where $q = |\mathcal{S}|/n$ is the sampling fraction. We then substitute this value into Equation (5) and solve for d:

$$\begin{aligned} d &= d_{\mathcal{S}} + d(1-q)(h_1/|\mathcal{S}|) \\ \hat{d}_{\text{uj1}} &= \big(1-(1-q)(h_1/|\mathcal{S}|)\big)^{-1} d_{\mathcal{S}}. \end{aligned} \tag{6}$$

For the sake of a self-contained presentation, we also include the definitions of $\hat{d}_{\text{uj2}}$, $\hat{d}_{\text{uj2a}}$, and $\hat{d}_{\text{Sh3}}$ but refer

to [37] for a detailed derivation and analysis of these estimators. The estimator $\hat{d}_{\text{uj2}}$ is defined as follows:

$$\hat{d}_{\text{uj2}} = \big(1-(1-q)(h_1/|\mathcal{S}|)\big)^{-1} \times \Big(d_{\mathcal{S}} - h_1(1-q)\ln(1-q)\hat{\gamma}^2(\hat{d}_{\text{uj1}})/q\Big), \tag{7}$$

which explicitly takes into account the variability of tuple frequencies. The estimator $\hat{d}_{\text{uj2a}}$ is a "stabilized" version of $\hat{d}_{\text{uj2}}$ that is obtained by removing any tuple whose frequency in the sample exceeds a fixed value c. Then $\hat{d}_{\text{uj2}}$ is computed over the remaining data and is incremented by the number of removed tuples. Finally, the modified Shlosser estimator is given by

$$\hat{d}_{\text{Sh3}} = d_{\mathcal{S}} + h_1 \left(\frac{\sum_{i=1}^{|\mathcal{S}|} iq^2(1-q^2)^{i-1}h_i}{\sum_{i=1}^{|\mathcal{S}|}(1-q)^i\big((1+q)^i-1\big)h_i}\right) \times \left(\frac{\sum_{i=1}^{|\mathcal{S}|}(1-q)^i h_i}{\sum_{i=1}^{|\mathcal{S}|} iq(1-q)^{i-1}h_i}\right)^2 \tag{8}$$

which assumes that $E[h_i]/E[h_1] \approx H_i/H_1$, where H_j is the number of tuples having a population frequency j.

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 $C_i = \sum_{\mathbf{t}_{ij}\in\mathcal{T}_i^o} f_{ij}/n$. The usual estimator of coverage, originally due to Turing (see [35]), is

$$\hat{C}_i = \max\big(1 - N_i^{(1)}/|\mathcal{S}|, |\mathcal{S}|/n\big). \tag{9}$$

This estimator assumes a frequency of one for unseen tuples, computing the coverage as one minus the fraction of singletons $N_i^{(1)}$ in the sample—that is the number of tuples that appear exactly once in $\mathcal{S}$ (i.e., $N_i^{(1)} = h_1$). We add the lower sanity bound $|\mathcal{S}|/n$ to handle the special case $N_i^{(1)} = |\mathcal{S}|$. For simplicity, we assume equal frequencies for all unobserved tuples. The resulting frequency estimation formula for tuple $\mathbf{t}_{ij}$ is

$$\hat{f}_{ij} = \begin{cases} (n/|\mathcal{S}|)\hat{C}_i F_{ij} & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^o \\ n(1-\hat{C}_i)/\hat{d}_i^u & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^u. \end{cases} \tag{10}$$

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 ($N_{ijk} \le l_k$). Set $Y_{ijk} = 1$ if $N_{ijk} > 0$ and $Y_{ijk} = 0$ otherwise. Then we estimate b_{ij} by its expected value $E[b_{ij}]$ under our maxEnt model:

$$\begin{aligned} \hat{b}_{ij} = E[b_{ij}] &= E\big[|\mathcal{B}_{ij}| + \sum_{k\in\mathcal{B}\setminus\mathcal{B}_{ij}} Y_{ijk}\big] \\ &= |\mathcal{B}_{ij}| + \sum_{k\in\mathcal{B}\setminus\mathcal{B}_{ij}} P(N_{ijk} > 0) \\ &= |\mathcal{B}_{ij}| + \sum_{k\in\mathcal{B}\setminus\mathcal{B}_{ij}} [1 - h(l_k, \hat{f}_{ij}^u, l)], \end{aligned} \tag{11}$$

where $h(a,b,c) = \binom{c-b}{a}\big/\binom{c}{a}$ 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 Equation (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 given by Equation (9). 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. 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 8. 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

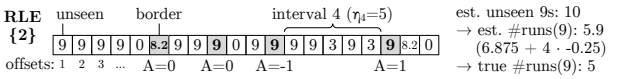


Fig. 8 Estimating the Number of RLE Runs $\hat{r}_{ij}$.

of unobserved $\mathbf{t}_{ij}$ instances assigned to the kth interval is hypergeometric, and we estimate f^u_{ijk} by its mean value: $\hat{f}^u_{ijk} = (\eta_k/\eta)\hat{f}^u_{ij}$. Given that $\hat{f}^u_{ijk}$ instances of $\mathbf{t}_{ij}$ are assigned randomly and uniformly among the η_k possible positions in the interval, the number of runs r_{ijk} within the interval (ignoring the borders) is known to follow a so-called "Ising-Stevens" distribution [46, pp. 422-423] and we estimate r_{ijk} by its mean: $\hat{r}_{ijk} = \hat{f}^u_{ijk}(\eta_k - \hat{f}^u_{ijk} + 1)/\eta_k$. 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:

$$\begin{aligned}\hat{A}_{ijk} &= E[A_{ijk}] \\ &\approx \left(\frac{\eta_k - \hat{f}^u_{ijk}}{\eta_k}\right)\left(\frac{\eta_{k+1} - \hat{f}^u_{ij(k+1)}}{\eta_{k+1}}\right)(1) \\ &\quad + \left(\frac{\hat{f}^u_{ijk}}{\eta_k}\right)\left(\frac{\hat{f}^u_{ij(k+1)}}{\eta_{k+1}}\right)(-1) \\ &= 1 - (2\hat{f}^u_{ijk}/\eta_k) \;=\; 1 - (2\hat{f}^u_{ij}/\eta).\end{aligned} \tag{12}$$

We modify this formula appropriately for the left- and rightmost borders. Our final estimate for the number of runs is $\hat{r}_{ij} = \sum_k \hat{r}_{ijk} + \sum_k \hat{A}_{ijk}$.

4.2 Partitioning Columns into Groups

Partitioning compressible columns into co-coded column groups comprises two major steps: column partitioning and column grouping. Column partitioning divides a given set of compressible columns into independent partitions in order to reduce the grouping costs. Column grouping then considers disjoint combinations of columns per partition. The overall objective of both steps is to maximize the compression ratio. Since exhaustive and brute-force grouping are infeasible, we focus on inexact but fast techniques.

Column Partitioning: We observed empirically that (1) column grouping usually generates groups of few columns, and that (2) the time needed for group extraction from the sample, to estimate its 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 divide the columns into a set of independent partitions and then apply grouping within each partition to form the column groups. Due to the super-linear complexity of grouping, partitioning can significantly reduce the overall costs. In detail, we provide two heuristic techniques:

- *Static Partitioning:* Correlated columns often appear in close proximity to each other. Static partitioning exploits this by dividing a list of columns into $\lceil m/k \rceil$ consecutive column partitions.
- *Bin Packing:* Since data characteristics affect grouping costs, we also provide a bin-packing-based technique. The weight of the i^{th} column is the cardinality ratio $\hat{d}_i/n$, indicating its estimated contribution to the grouping costs. The capacity of a bin is a tuning parameter β, which ensures moderate grouping costs. Bin packing minimizes the number of bins, which maximizes grouping potential while controlling the processing costs. We made the design choice of a constant bin capacity—independent of z_i—to ensure constant compression throughput irrespective of blocking configurations. We solve this problem with the first-fit decreasing heuristic [45].

Column Grouping: A brute-force greedy method for grouping a set of compressible columns into column groups starts with singleton groups and executes merging iterations. At each iteration, we merge the two groups yielding maximum compression ratio with regard to the entire block, i.e., minimum absolute change in size $\Delta\hat{S}_{ij} = \hat{S}_{ij} - \hat{S}_i - \hat{S}_j$. We terminate when no further size reductions are possible, i.e., no change in size is below 0. Although compression ratios are estimated from a sample, the cost of the brute-force method is $O(m^3)$. Our greedy column grouping algorithm (Algorithm 2) improves this naïve brute-force method via pruning and memoization. We execute merging iterations until the working set W does not change anymore (lines 2-14). In each iteration, we enumerate all $|W|\cdot(|W|-1)/2$ candidate pairs of groups (lines 4-5). A candidate can be safely pruned if any of its input groups has a size smaller than the currently best change in size $\Delta\hat{S}_{\text{opt}}$, i.e., the change in size of the best group opt (lines 7-8). This pruning threshold uses a natural lower bound $\underline{\hat{S}}_{ij} = \max(\hat{S}_i, \hat{S}_j)$ because at best the smaller group does not add any size. Substituting $\underline{\hat{S}}_{ij}$ into $\Delta\hat{S}_{ij}$ yields the lower bound $\Delta\underline{\hat{S}}_{ij} = -\min(\hat{S}_i, \hat{S}_j)$. Although

Algorithm 2 Greedy Column Grouping

Input: A set of columns C
Output: A set of column groups CG

```
1: W' ← C, W ← ∅
2: while W' ≠ W do                    // until no further compression
3:   W ← W', opt ← null
4:   for all i in 1 to |W| do
5:     for all j in i+1 to |W| do
6:       // Candidate pruning (without group extraction)
7:       if −min(Ŝ_{W_i}, Ŝ_{W_j}) > ΔŜ_opt then
8:         continue
9:       // Group creation with memoization
10:      W_ij ← createGroup(W_i, W_j, memo)
11:      if ΔŜ_{W_ij} < ΔŜ_opt then
12:        opt ← W_ij
13:  if ΔŜ_opt < 0 then                 // group selection
14:    W' ← W ∪ opt − opt_i − opt_j     // modify working set
15: return CG ← W'
```

this pruning does not change the worst-case complexity, it works very well in practice: e.g., on Census, we prune 2,789 of 3,814 candidates. Any remaining candidate is then evaluated, which entails extracting the column group from the sample, estimating its size $\hat{S}$, and updating the best group opt (lines 10-12). Observe that each merging iteration enumerates $O(|W|^2)$ candidates, but—ignoring pruning—only $O(|W|)$ candidates have not been evaluated in prior iterations; these are the ones formed by combining the previously merged group with each element of the remaining working set. Hence, we apply memoization in createGroup to reuse computed statistics such as $\hat{S}_{ij}$, which reduces the overall worst-case complexity—in terms of group extractions—from $O(m^3)$ to $O(m^2)$. On Census, we reuse 501 of the remaining 1,025 candidates. Finally, we select the merged group and update the working set (lines 13-14). Maintaining opt within the memo table across iterations can improve pruning efficiency but requires removing overlapping groups upon group selection.

4.3 Compression Algorithm

We now describe the matrix block compression algorithm (Algorithm 3). Note that we transpose the input in case of row-major dense or sparse formats to avoid performance issues due to column-wise extraction.

Planning Phase (lines 2-12)**:** Planning starts by drawing a sample of rows $\mathcal{S}$ from $\mathbf{X}$. For each column i, the sample is first used to estimate the compressed column size S_i^{C} by $\hat{S}_i^{\mathrm{C}} = \min(\hat{S}_i^{\mathrm{RLE}}, \hat{S}_i^{\mathrm{OLE}}, \hat{S}_i^{\mathrm{DDC}})$, where $\hat{S}_i^{\mathrm{RLE}}$, $\hat{S}_i^{\mathrm{OLE}}$, and $\hat{S}_i^{\mathrm{DDC}}$ are obtained by substituting the estimated $\hat{d}_i$, $\hat{z}_i$, $\hat{r}_{ij}$, and $\hat{b}_{ij}$ into formulas (1)–(3). We conservatively estimate the uncompressed column size as $\hat{S}_i^{\mathrm{UC}} = \min(n\alpha, \hat{z}_i(4+\alpha))$, which covers both dense and sparse, with moderate underestimation for sparse as it ignores CSR row pointers. However,

Algorithm 3 Matrix Block Compression

Input: Matrix block $\mathbf{X}$ of size $n \times m$
Output: A set of compressed column groups $\mathcal{X}$

```
1: C^C ← ∅, C^UC ← ∅, G ← ∅, X ← ∅
2: // Planning phase - - - - - - - - - - - - - - - - - - -
3: S ← sampleRowsUniform(X, sample_size)
4: for all columns i in X do                    // classify
5:   cmp_ratio ← ẑ_i α / min(Ŝ_i^RLE, Ŝ_i^OLE, Ŝ_i^DDC)
6:   if cmp_ratio > 1 then
7:     C^C ← C^C ∪ i
8:   else
9:     C^UC ← C^UC ∪ i
10: bins ← runBinPacking(C^C)                   // group
11: for all bins b in bins do
12:   G ← G ∪ greedyColumnGrouping(b)
13: // Compression phase - - - - - - - - - - - - - - - - -
14: for all column groups G_i in G do           // compress
15:   do
16:     biglist ← extractBigList(X, G_i)
17:     cmp_ratio ← getExactCmpRatio(biglist)
18:     if cmp_ratio > 1 then
19:       X ← X ∪ compressBigList(biglist), break
20:     k ← removeLargestColumn(G_i)
21:     C^UC ← C^UC ∪ k
22:   while |G_i| > 0
23: return X ← X ∪ createUCGroup(C^UC)
```

this estimate 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 our greedy column grouping (Algorithm 2) per 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 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 groups are extracted during a single column-wise pass through $\mathbf{X}$ using hashing. During this extraction operation, the parameters d_i, z_i, r_{ij}, and b_{ij} for each group $\mathcal{G}_i$ are computed exactly, with negligible overhead. These parameters are used in turn to calculate the exact compressed sizes S_i^{OLE}, S_i^{RLE}, and S_i^{DDC} 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 groups as part of a single 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 format 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)–(3). The incompressible columns are collected into a single UC column group that is encoded in sparse or dense format based on the exact number of non-zeros.

Parallel Compression: Our compression algorithm also allows for straightforward parallelization. For multi-threaded compression, we simply apply a multi-threaded transpose and replace the sequential `for` loops on lines 4, 11, and 14 with parallel `parfor` loops because these loops are free of loop-carried dependencies. For distributed compression, we retain a blocked matrix representation and compress matrix blocks independently in a data-local manner.

5 Experiments

We study CLA in SystemML over a variety of ML programs and real-world datasets in order to understand compression characteristics and operation performance. To summarize, the major insights are:

Operations Performance: CLA achieves in-memory matrix-vector multiply performance close to uncompressed operations without need for decompression. 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 9.2x and 2.6x, respectively, when uncompressed or lightweight-compressed matrices do not fit in memory.

Effective Compression Planning: Sampling-based compression planning yields good compression plans—i.e., good choices of encoding formats and co-coding schemes, and thus, good compression ratios—at moderate 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 enabled 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 enabled, 96 GB RAM @1.33 GHz (ECC, registered), 12x2 TB disks, 10Gb Ethernet, and CentOS Linux 7.3. The nominal peak performance per node for memory bandwidth and floating point operations are 2x32 GB/s from local memory (we measured 47.9 GB/s), 2x12.8 GB/s over QPI (Quick Path Interconnect), and 2x115.2 GFLOP/s. We used OpenJDK 1.8.0, and Apache Hadoop 2.7.3, configured with 11 disks for HDFS and local working directories. We ran Apache Spark 2.1, in yarn-client mode, with 6 executors, 25 GB driver memory, 60 GB executor memory, and 24 cores per executor. Finally, we used Apache SystemML 0.14 (RC1, April 2017) with default configuration, except for a larger block size of 16K rows.

Table 4 ULA/CLA Default Parameters.

Parameter	Value
Matrix block size	16,384
Sparsity threshold $\text{nnz}/(n \cdot m)$	$\text{nnz}/(n \cdot m) < 0.4 \wedge S^{\text{UC}}_{\text{sparse}} < S^{\text{UC}}_{\text{dense}}$
Sample fraction q	0.05
Hybrid estimator α_1/α_2	0.9 / 30
Column partitioning	Bin Packing
Bin capacity β	$3.2 \cdot 10^{-5}$
OLE/RLE cache block size Δ^c	$2\Delta^s = 2^{17}$
DDC cache block size	2,048

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 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 block including compression planning. Making our compressed matrix block a subclass of the uncompressed matrix block yielded seamless integration of all operations, serialization, buffer pool interactions, and dynamic recompilation. In case of unsupported operations, we automatically decompress and apply uncompressed operations, which was not necessary in our end-to-end experiments.

Baseline Comparisons: To isolate the effects of compression, we compare CLA against Apache SystemML 0.14 with (1) uncompressed linear algebra (ULA), which uses either sparse or dense Java linear algebra operations[2], (2) heavyweight compression (Gzip), and (3) lightweight compression (Snappy and LZ4[3]), where we use native compression libraries and ULA. We also compare with (4) CSR-VI [52], a sparse format with dictionary encoding, and D-VI, a derived dense format. Table 4 shows the general as well as CLA-specific configuration parameters used throughout

[2] The results with native BLAS libraries would be similar because memory-bandwidth and I/O are the bottlenecks.

[3] For consistency with previously published results [32], we use Snappy, which was the default codec in Spark 1.x. However, we also include LZ4, which is the default in Spark 2.x.

Table 5 Compression Plans of Real Datasets (min/max counts over multiple runs, number of columns per type in parentheses).

Dataset	m	$\lvert\mathcal{X}\rvert$	#OLE	#RLE	#DDC1	#DDC2	#UC	#Values
Higgs	28	17	0	0	1 (4)	15 (15)	1 (9)	218,739
Census	68	14	4 (20)	7 (27)	2 (14)	0	0	16,563
	〃	16	5 (24)	9 (34)	3 (20)	〃	〃	22,719
Covtype	54	28	4 (4)	18 (43)	4 (4)	2 (2)	0	14,026
	〃	29	5 (5)	19 (44)	〃	〃	〃	14,027
ImageNet	900	604	425 (559)	0	170 (306)	0	0	95,002
	〃	627	457 (594)	〃	187 (341)	〃	〃	100,594
Mnist8m	784	767	578 (586)	0	198 (198)	0	0	187,195
	〃	781	583 (586)	〃	〃	〃	〃	200,815
Airline67	29	23	10 (10)	3 (4)	2 (3)	7 (8)	0	42,016
	〃	25	13 (13)	4 (7)	2 (4)	〃	〃	43,327

our experiments unless otherwise specified. The sparsity threshold indicates when we use a sparse uncompressed block representation and operations. For the hybrid estimator from Equation (4), we use the recommended threshold parameters of $\alpha_1 = 0.9$ and $\alpha_2 = 30$ [37], numerically stable implementations, and for $\hat{d}_{\text{uj2a}}$, a modified stabilization cut-off $c = \max_{\mathbf{t}_{ij} \in \mathcal{T}_i^o}(F_{ij})/2$ instead of $c = 50$ as it provided better robustness in our setting.

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 $\lambda = 10^{-3}$. LinregDS/PCA are non-iterative and LinregCG is the only iterative algorithm without nested loops. We ran all experiments over real and scaled real datasets, introduced in Table 1. For our end-to-end experiments, we used (1) the *InfiMNIST* data generator [19] to create an Mnist480m dataset of 480 million observations with 784 features (1.1 TB in binary format) and binomial, i.e., binary, class labels, as well as (2) replicated versions of the ImageNet dataset and again binomial labels. We use binomial[4] labels to compare a broad range of ML algorithms.

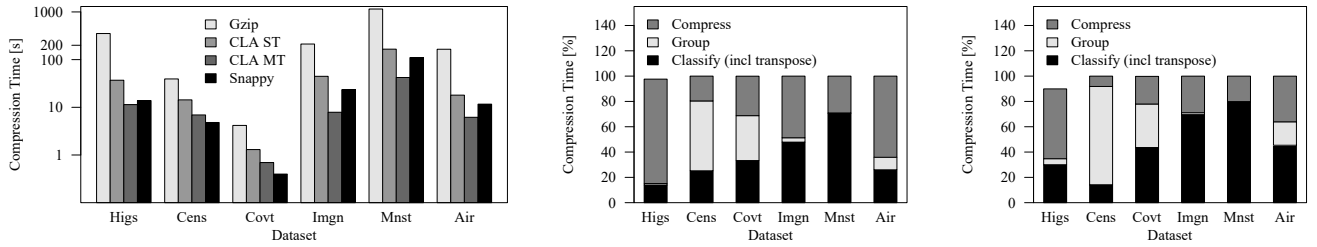
5.2 Compression and Operations

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.

Summary of Compression Plans: To explain the micro benchmarks on operations performance, we first summarize the compression layouts for our datasets. Due to sample-based compression planning with random seeds, there are moderate variations between layouts for different runs. However, the compressed sizes differed by less than 3.9% in all cases. Table 5 shows the layouts observed over 20 runs in terms of the number of column groups per type (with the number of columns in parentheses), as well as the total number of distinct tuples after co-coding; we report min and max counts as two rows—these coincide for Higgs. We see that OLE and DDC are more common than RLE. Both OLE and DDC are applied on almost every dataset, targeting complementary sparse and dense data and thereby handling non-uniform sparsity. Furthermore, we see that Higgs is the only dataset with an uncompressed column group, and that co-coding was applied on all datasets. The datasets Higgs, ImageNet, Mnist8m, and to some extend Airline78, show a large number of values, although we excluded the uncompressed group of Higgs. This is because we use local value dictionaries per column group, which store common values across groups redundantly. Compared to previous results [32], the new DDC column encoding format reduced the number of distinct tuples on Census, ImageNet, and Mnist8m.

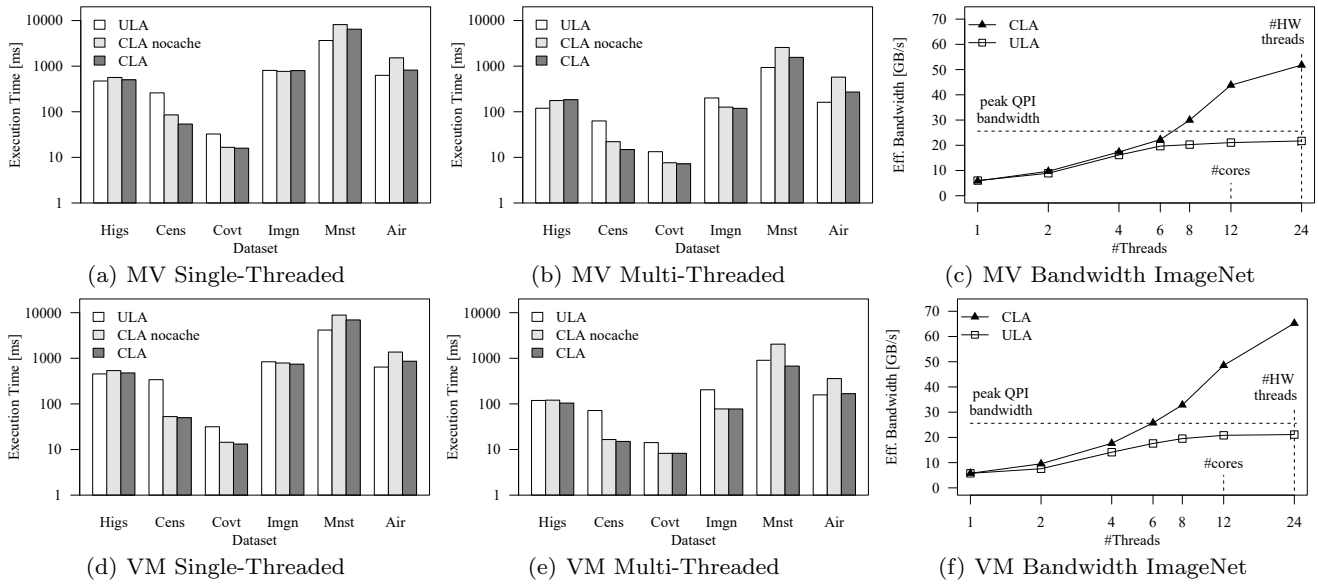
Compression: Figure 9 shows the times for creating a single compressed matrix block. Figure 9(a) shows the absolute compression times in log scale, where we see reasonable average bandwidth across all datasets of roughly 100 MB/s (ranging from 67.7 MB/s to 184.4 MB/s), single-threaded. In comparison, the

[4] For Mnist with its original 10 classes, we created the labels with $\mathbf{y} \leftarrow (\mathbf{y} == 7)$ (i.e., class 7 against the rest), whereas for ImageNet with its 1,000 classes, we created the labels with $\mathbf{y} \leftarrow (\mathbf{y}_0 > (\max(\mathbf{y}_0) - (\max(\mathbf{y}_0) - \min(\mathbf{y}_0))/2))$, where we derived $\mathbf{y}_0 = \mathbf{X}\mathbf{w}$ from the data $\mathbf{X}$ and a random model $\mathbf{w}$.



(a) Absolute Time (b) Time Breakdown CLA ST (c) Time Breakdown CLA MT

Fig. 9 Compression Time (Gzip single-threaded, CLA single- and multi-threaded (CLA ST/MT), Snappy single-threaded).



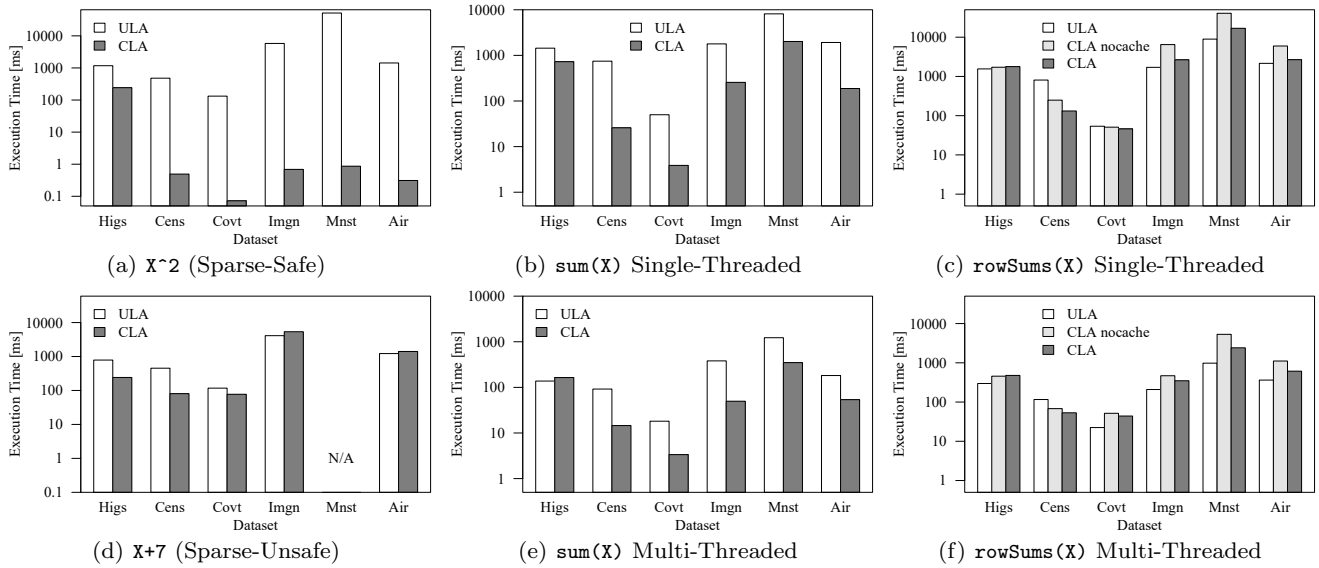
(a) MV Single-Threaded (b) MV Multi-Threaded (c) MV Bandwidth ImageNet

(d) VM Single-Threaded (e) VM Multi-Threaded (f) VM Bandwidth ImageNet

Fig. 10 Matrix-Vector (MV) and Vector-Matrix (VM) Multiplication Time and Bandwidth.

single-threaded compression throughput of the general-purpose Gzip and Snappy using native libraries, ranged from 6.9 MB/s to 35.6 MB/s and 156.8 MB/s to 353 MB/s, respectively. Figure 9(b) further shows the time breakdown of individual compression steps. Bar heights below 100% are due to the final extraction of uncompressed column groups. Depending on the dataset, any of the three compression steps (classification, column grouping, or the actual compression) can turn into bottlenecks. The classification time is largely influenced by the time for matrix transposition, especially for sparse datasets (i.e., Covtype, ImageNet, and Mnist8m) because sparse transpose is challenging with regard to its cache-conscious implementation. However, repeated row-wise extraction easily amortizes the transpose cost compared to column-wise extraction. We also report multi-threaded compression times, but only for CLA because Gzip or Snappy would require us to split the matrix into multiple blocks to allow multi-threaded compression and deserialization, parallelized over individual blocks. The observed speedups are reasonable, ranging from 2x to 7x which is largely affected by the ratio of compression time spent in column co-coding. Figure 9(b) shows the time breakdown for multi-threaded compression, showing that, for example, Census is affected by load imbalance of column grouping because we parallelize over bins, where the number of bins limits the maximum degree of parallelism.

Matrix-Vector Multiplication: Figure 10(a) and 10(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 or better than ULA, with two exceptions of Mnist8m and Airline78, where CLA performs significantly worse. This behavior is mostly caused by (1) a large number of values in OLE column groups which require multiple passes over the output vector, and (2) the size of the output vector. For Mnist8m (8M rows) and Airline67 (14M rows), the output 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 operations due to cache thrashing effects. ULA constitutes a competitive baseline because its multi-threaded implementation achieves peak single-socket or remote memory



(a) `X^2` (Sparse-Safe) (b) `sum(X)` Single-Threaded (c) `rowSums(X)` Single-Threaded

(d) `X+7` (Sparse-Unsafe) (e) `sum(X)` Multi-Threaded (f) `rowSums(X)` Multi-Threaded

Fig. 11 Matrix-Scalar and Unary Aggregate Operation Time.

bandwidth of $\approx$ 25 GB/s. Multi-threaded CLA operations show a speedup similar to ULA due to parallelization over logical row partitions, in some cases even better. For example, Figure 10(c) shows the effective memory bandwidth of matrix-vector multiplication on ImageNet. We see that with increasing number of threads, ULA quickly saturates peak single-socket or remote QPI memory bandwidth (because without NUMA-awareness the JVM allocates the matrix locally on the socket of the current thread). In contrast, CLA achieves almost a 2x improvement with 24 threads due to smaller bandwidth requirements and because multi-threading mitigates any additional overheads.

Vector-Matrix Multiplication: Figures 10(d) and 10(e) 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 Mnist8m and Airline78, due to large input vectors that exceed the L3 cache size and repeated scattered scans of these vectors for many values. However, cache-conscious CLA operations overcome this issue and achieve substantial improvements, especially in case of multi-threaded operations. ULA is again a strong baseline at peak single-socket or remote memory bandwidth. It is noteworthy that CLA largely benefits from our post-scaling technique by avoiding false sharing, which caused slowdowns by more than an order of magnitude on some of our datasets. As with matrix-vector products but to an even greater degree, multi-threaded CLA vector-matrix operations exhibit greater speedup than ULA because ULA becomes memory-bandwidth bound, whereas CLA has less bandwidth requirements due to smaller size in compressed form. For example, Figure 10(f) shows the effective bandwidth on ImageNet, with varying number of threads, where CLA exceeds the peak single-socket/remote memory bandwidth of 25 GB/s by more than 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 the 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 or tuple references, whereas ULA has to compute every non-zero entry. We see improvements of three to five orders of magnitude, except for Higgs which contains a large uncompressed group with 9 out of 28 columns. Figure 11(d) shows the results for the sparse-unsafe `X+7`, where CLA and ULA perform similarly because CLA has to materialize modified offset lists that include added and removed values. The improvements on Higgs and Census are due to a large fraction of columns being encoded as DDC columns groups. DDC computes sparse-unsafe operations like `X+7` purely over distinct value tuples because zeros are represented in this dense encoding, eliminating the need to add or remove tuples. Finally, Mnist8m is not applicable here (N/A) because SystemML's dense matrix blocks are limited to 16 GB.

Unary Aggregate Operations: Figures 11(b) and 11(e) 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, or counting of DDC references—we see improvements of up to 1.5 orders of magnitude compared to ULA, which is at peak memory bandwidth. The only exception is Higgs due to its large uncompressed column group. Multi-threaded CLA operations show

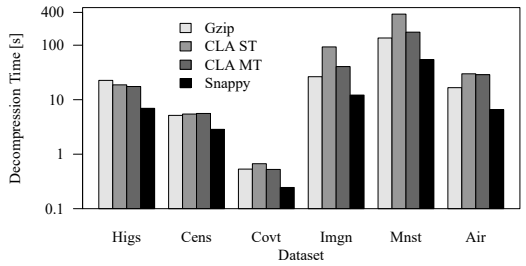


Fig. 12 Decompression Time (Gzip and Snappy single-threaded, CLA single- and multi-threaded (CLA ST/MT)).

runtime improvements up to an order of magnitude, slightly worse than ULA. Furthermore, Figures 11(c) and 11(f) show the single- and multi-threaded aggregation time for `rowSums(X)`, which is more challenging on compressed column groups as it does not allow for efficient counting per value and the output is a two column matrix for row sums and corrections. These corrections are necessary because `sum(X)` and `rowSums(X)` use a numerically stable Kahan addition [86]. However, we see performance close to ULA operations, due to pre-aggregation of co-coded tuples as well as cache-conscious operations (with a modified cache-partition size of $\Delta^c = \Delta^s$), all of which contributed to speedups of up to 3x compared to basic CLA operations.

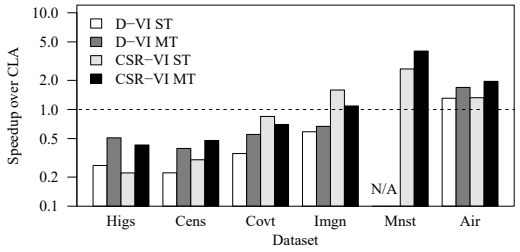
Decompression: In the rare case of unsupported operations, we decompress and perform ULA operations. In contrast, Gzip or Snappy need to decompress block-wise for *every* operation. Figure 12 shows the single- and multi-threaded CLA decompression time compared to single-threaded Gzip and Snappy decompression and deserialization. Multi-threaded general-purpose decompression would again require parallelization over multiple matrix blocks. Overall, we see passable CLA decompression time similar to heavy-weight Gzip. Multi-threaded decompression achieves only moderate speedups because decompression is bound by allocation and write memory bandwidth. On sparse datasets, we benefit from multi-threading due to latency hiding with speedups of up to 2.5x.

5.3 Comparison to CSR-VI

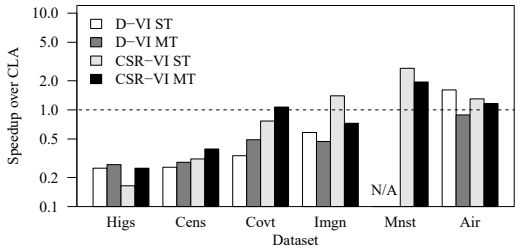
CSR-VI (CSR Value Indexed) [52] is prior work on *lossless* matrix value compression via dictionary encoding. We compare CSR-VI and a derived dense format that we call D-VI, both of which use implementations for 1, 2, and 4-byte codes. These formats—especially D-VI—are very similar to our DDC encoding format with the differences of a row-major instead of column-major representation, and a single dictionary for the entire matrix as opposed to a dictionary per column group.

Table 6 Compression Ratio CSR-VI/D-VI vs. CLA.

Dataset	Sparse	#Values	CSR-VI	D-VI	CLA
Higgs	N	8,083,944	1.04	1.90	**2.17**
Census	N	46	3.62	7.99	**35.69**
Covtype	Y	6,682	3.56	2.48	**18.19**
ImageNet	Y	824	2.07	1.93	**7.34**
Mnist8m	Y	255	2.53	N/A	**7.32**
Airline78	N	7,795	1.77	3.99	**7.44**



(a) Matrix-Vector Multiplication Time

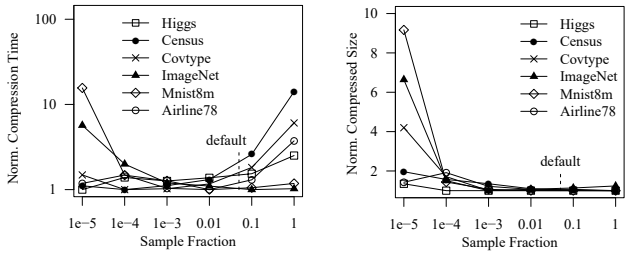


(b) Vector-Matrix Multiplication Time

Fig. 13 Performance CSR-VI/D-VI (single- and multi-threaded, i.e., ST/MT) vs. CLA ST/MT.

Compression Ratio: Table 6 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. Due to the single dictionary for the entire CSR-VI or D-VI matrix block, high-cardinality columns negatively affect the compression ratios of other low-cardinality columns. At the same time, however, CSR-VI and D-VI benefit from this shared dictionary—in terms of faster lookups due to the smaller dictionary size—on datasets like Mnist8m, where columns exhibit many common distinct values.

Operations Performance: Figure 13 further shows the single- and multi-threaded (ST/MT) 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 generally better. Mnist8m and Airline are exceptions where CSR-VI and D-VI always perform better because of compact



(a) Compression Time (b) Compressed Size

Fig. 14 Influence of Sample Fraction.

1- and 2-byte encodings as well as the shared dictionary of common values which fit into L1 and L2 cache, respectively. In contrast, CLA uses disjoint dictionaries across column groups and many values as shown in Table 5. In comparison to previous results [32], we see substantial CLA improvements, especially for datasets like Higgs, Census, and partially Mnist8m.

Overall, CLA shows operations performance similar or better than CSR-VI/D-VI, 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 the sensitivity of CLA parameters—with regard to the used sample fraction and bin capacity for co-coding decisions—as well as the size estimation accuracy, where we report the average of 5 runs.

Sample Fraction: Sampling-based compression planning is crucial for fast compression. Figure 14 shows the compression time and compressed size (minimum-normalized) with varying sample fraction. We see large time improvements of more than an order of magnitude using small sample fractions. Note that we report total compression time that includes overheads such as transposing the input matrix, which is not affected by the sample fraction. 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 9(b)). Furthermore, Higgs is the only dataset where the best compression time was obtained for the smallest sample fraction of 10^{-5} because estimation errors led to a larger uncompressed group, which was cheaper to extract. By default, we use a very conservative, low-risk sample fraction of 0.05 instead of 0.01 in previously published results [32]. Our new greedy column grouping algorithm (see Algorithm 2) and various runtime improvements enabled the larger sample fraction without sacrificing compression performance.

Table 7 Size Estimation Accuracy (Average ARE), with sample fractions $q = 0.01$ and $q = 0.05$.

Fraction q	Dataset	Excerpt [25]	CLA Est.
0.01	Higgs	**9.4%**	17.4%
	Census	171.4%	**3.8%**
	Covtype	60.6%	**11.5%**
	ImageNet	21.4%	**6.7%**
	Mnist8m	6.7%	**1.5%**
	Airline78	**5.3%**	10.1%
0.05	Higgs	**2.1%**	13.7%
	Census	159.2%	**0.5%**
	Covtype	35.4%	**11.8%**
	ImageNet	19.3%	**1.9%**
	Mnist8m	**1.0%**	1.2%
	Airline78	**3.0%**	4.1%

Bin Weights: Our bin-packing-based column group partitioning reduces the number of candidate column groups with bin capacity β, which is a tuning parameter. The larger β, the larger the partitions on which we apply greedy column grouping. Large partitions likely lead to more co-coding, i.e., a smaller number of groups, but also larger compression time, especially, for datasets with many columns like ImageNet or Mnist8m. We varied $\beta = 3.2x$ with $x \in \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}, 10^{-8}, 10^{-9}\}$ and observed that our default of $\beta = 3.2 \cdot 10^{-5}$ achieves a good tradeoff between compression time and compressed size, yielding compressed sizes within 18.6% (often much less) of the observed minimum.

Estimation Accuracy: We compare our CLA size estimators (using sample fractions of $q = 0.01$ and $q = 0.05$) with a systematic Excerpt [25] (using the first $0.01n$ and $0.05n$ rows, respectively) that allows us to observe compression ratios even for challenging formats such as RLE. Table 7 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 on average significantly better accuracy due to robustness against skew and effects of value tuples. Furthermore, we see a systematic reduction in ARE with increased sample fraction for both Excerpt and CLA, but with large variations of relative improvements. Datasets with DDC2 groups in the final plans—i.e., Higgs, Covtype, and Airline78—show generally higher errors because DDC can be difficult to predict. This is due to a hard cut-off point between DDC1 and DDC2 at 256 distinct values, which affects compressed size by almost 2x. Finally, Excerpt also resulted in worse plans because column grouping mistakes could not be corrected.

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

Table 8 Mnist8m Deserialized RDD Storage Size.

Block Size	ULA	Snappy	LZ4	CLA
1,024	18 GB	7.4 GB	7.1 GB	7.9 GB
2,048	〃	〃	〃	5.6 GB
4,096	〃	〃	〃	4.8 GB
8,192	〃	〃	〃	3.8 GB
16,384	**18 GB**	**7.4 GB**	**7.1 GB**	**3.2 GB**

Table 9 ImageNet Deserialized RDD Storage Size.

Block Size	ULA	Snappy	LZ4	CLA
1,024	3.9 GB	1.2 GB	1.3 GB	0.9 GB
2,048	〃	〃	〃	0.8 GB
4,096	〃	〃	〃	0.7 GB
8,192	〃	〃	〃	0.6 GB
16,384	**3.9 GB**	**1.2 GB**	**1.3 GB**	**0.6 GB**

5.5 End-to-End Experiments

To study the end-to-end CLA benefits, we ran several algorithms over subsets of Mnist480m and scaled versions of ImageNet. We report the end-to-end runtimes as the average of 3 runs, including read from HDFS, Spark context creation, and on-the-fly compression. The baselines are ULA and Spark's RDD compression with Snappy and LZ4, respectively.

RDD Storage: ULA and CLA use the deserialized storage level `MEM_AND_DISK`, while Snappy and LZ4 use `MEM_AND_DISK_SER` because RDD compression requires serialized data. ULA further uses a compact CSR format instead of our default Modified-CSR format for sparse matrix blocks that are cached in aggregated memory [16]. However, both sparse block formats have the same serialized representation. Tables 8 and 9 show the RDD storage size of Mnist8m and ImageNet with varying SystemML block size. For 16K, we observe compression ratios of 2.4x and 3.3x for Snappy but 5.6x and 6.5x for CLA, which is a moderate improvement over previous results [32]. LZ4 shows compression ratios similar to Snappy. In contrast to these general-purpose schemes, CLA's compression advantage increases with larger block sizes because the common header—including the dictionaries—is stored only once per column group per block. For these image datasets, CLA would further benefit from a shared dictionary across column groups, which we do not apply for the sake of simple compression and serialization. We obtained similar ratios for larger subsets of Mnist480m and scaled versions of ImageNet.

L2SVM on Mnist: We first investigate L2SVM as a common classification algorithm. Given the described setup, we have a maximum aggregated memory of $6 \cdot 60\,\text{GB} \cdot 0.6 = 216\,\text{GB}$, which might be reduced to $6 \cdot 60\,\text{GB} \cdot 0.5 = 180\,\text{GB}$ depending on required execution memory. 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 15 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 30% and 4%, 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 us 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.

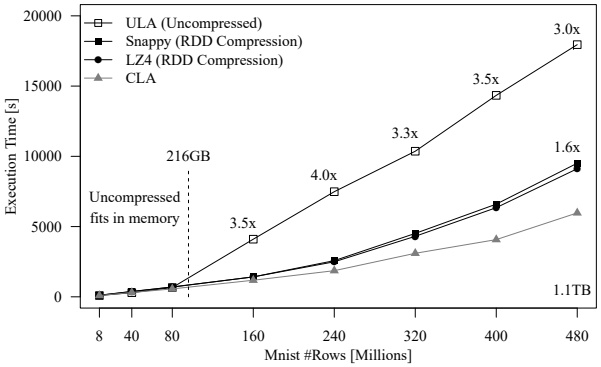


Fig. 15 L2SVM End-to-End Performance Mnist.

Comparison to Previous Results: In comparison to previously published results [32], ULA and Snappy show significantly better runtimes. This is primarily due to improvements in Spark 2.1 and SystemML 0.14. Specifically, this includes Spark's unified memory management [67] (since Spark 1.6), which reduces spilling and garbage collection, as well as improvements for memory efficiency in SystemML. Both aspects primarily affect out-of-core scenarios and thus, only ULA and Snappy. In contrast, the improvements of CLA are mostly due to the extensions described in this paper, which has been validated with Spark 1.5.

Other ML Algorithms on Mnist: We further 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 10 shows the results for the interesting points of Mnist40m (90 GB), where all datasets fit in memory, as well as Mnist240m (540 GB) and Mnist480m (1.1 TB),

Table 10 ML Algorithms End-to-End Performance Mnist40m/240m/480m.

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

Table 11 ML Algorithms End-to-End Performance ImageNet15/150.

Algorithm	**ImageNet15** (65 GB)			**ImageNet150** (650 GB)		
	ULA	**Snappy**	**CLA**	**ULA**	**Snappy**	**CLA**
L2SVM	191 s	241 s	**188 s**	8,631 s	2,719 s	**1,794 s**
Mlogreg	283 s	420 s	**266 s**	25,361 s	5,257 s	**2,994 s**
GLM	207 s	293 s	**195 s**	21,272 s	4,324 s	**2,307 s**
LinregCG	**78 s**	110 s	82 s	3,012 s	1,013 s	**683 s**
LinregDS	**71 s**	167 s	132 s	1,530 s	1,491 s	**1,181 s**
PCA	**80 s**	172 s	131 s	1,337 s	1,451 s	**1,148 s**

where uncompressed datasets no longer fit in memory and Snappy- and CLA-compressed datasets reach their respective in-memory limits. For Mnist40m and iterative algorithms, we see similar ULA/CLA performance but a slowdown of up to 57% with Snappy. This is because RDD compression incurs decompression overhead per iteration, whereas CLA's initial compression cost is amortized over multiple iterations. For non-iterative algorithms, CLA and Snappy show overheads of up to 92% and 87%, respectively. Beside the initial compression overhead, CLA also shows less efficient TSMM performance, while the RDD decompression overhead, is mitigated by initial read costs. For Mnist240m and Mnist480, we see significant performance improvements by CLA—of up to 7.9x and 2.6x—compared to ULA and RDD compression for Mlogreg, GLM, and LinregCG. This is due to many inner iterations with RDD operations in the outer and inner loops. Note that all three algorithms use one RDD operation for a matrix-vector chain per inner iteration. Finally, for LinregDS and PCA, CLA shows again inferior TSMM performance but slight improvements over ULA and Snappy.

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 ratios reported in Table 9. To summarize, Table 11 shows the results for ImageNet15 (65 GB), where all datasets fit in memory, and ImageNet150 (650 GB). For LinregDS and PCA, CLA performs slightly better than on Mnist due to better matrix-vector and vector-matrix and thus TSMM performance (see Figures 10(c) and 10(f)). Overall, we see similar results with improvements of up to 9.2x and 1.8x, respectively.

Table 12 L2SVM End-to-End Performance Mnist (with code generation for operator fusion of cell-wise operations).

Dataset	Codegen disabled		Codegen enabled	
	ULA	**CLA**	**ULA**	**CLA**
Mnist40m	**296 s**	308 s	**173 s**	181 s
Mnist240m	7,483 s	**1,861 s**	6,695 s	**1,068 s**
Mnist480m	17,950 s	**5,973 s**	14,381 s	**3,565 s**

Effect of Code Generation: Given the improvements of ULA and Snappy induced by Spark 2.1 and SystemML 0.14 compared to previous results, it is important to understand the remaining bottlenecks. We recently added code generation as an experimental feature in SystemML 0.14 [31], which has the potential to reduce materialized intermediates, reduce scans of matrices, and exploit sparsity across operations. Table 12 compares the L2SVM end-to-end performance on Mnist for ULA and CLA with and without code generation[5]. We observe that code generation yields constant absolute improvements for both ULA and CLA, which increase with increasing data size. This is due to $O(n)$ vector intermediates. For example, on Mnist480m, each of these vectors is already 3.8 GB large, leading to unnecessary overhead, which includes writes to and reads from main memory as well as bufferpool evictions. Code generation significantly reduces the number of these vector intermediates (for example, for L2SVM's inner loop from four to zero). Similar to Amdahl's law, this overhead constitutes the "serial fraction" which limits the speedup of compression. Accordingly, with code generation enabled, the speedup increased from 4x to 6.2x for Mnist240m. We leave a deeper analysis of these

[5] We enabled code generation for cell-wise operations only because SystemML 0.14 does not yet support operator fusion, i.e., code generation, for compressed matrices.

effects and operator fusion for compressed matrices as an interesting direction for future work.

Overall, CLA shows positive results with significant improvements for iterative algorithms due to smaller memory bandwidth requirements and reduced I/O. Note, however, that the achieved speedups are directly influenced by the obtained compression ratios and hence, strongly depend on the given dataset.

6 Discussion of Open Issues

Compressed linear algebra shows very promising results. In this section, we further discuss current limitations and aspects that are beyond the scope of this paper, some of which are directions for future work.

Toward Global Planning: So far, we applied local—i.e., per matrix block—compression planning. The advantages are simplicity and efficiency because blocks can be processed independently. However, there are various opportunities for global planning. First, the global block size affects compression ratios but also memory requirements per block operation and the degree of parallelism. Second, we could exploit the knowledge of program-wide operations to decide upon compression schemes. For example, value-based operations such as $\max(\mathbf{X}, 0)$ motivate formats with value-row mapping such as OLE and RLE, whereas other operations such as indexing motivate schemes with row-value mapping such as DDC. Both decisions require the extraction of pertinent global summary statistics and their inclusion into compilation decisions. We leave this as an interesting direction for future work.

Design Space of Compression Schemes: The design space of alternative compression schemes is large, with tradeoffs between compression ratio, compression overhead, and operation performance. We aimed at good compression ratios with operation performance close to, or better than, the uncompressed case. Specifically, we used compressed offset lists inspired by sparse matrix representations, bitmap compression, and dictionary coding, inspired by dictionary coding in column stores. A systematic evaluation of this design space with regard to alternative compression techniques is interesting future work. In the presence of integer data—for example, originating from encoded categorical or binned data—we could even apply very lightweight compression techniques [27] such as null-suppression or patched frame-of-reference.

Operations beyond Matrix-Vector: So far, we mostly discussed quasi-unary operations of a single compressed matrix block with uncompressed vectors which covers a large class of ML algorithms. The advantage is simplicity because operations can be composed of independent column group operations. However, unsupported operations require decompression. An interesting research question—beyond the scope of this paper—is how to realize efficient matrix-matrix operations in the face of a heterogeneous collection of uncompressed sparse or dense matrix blocks, together with compressed blocks having heterogeneous encodings. As shown in Section 3.3, the opportunity is a potential reduction in the number of floating point operations—especially in the presence of co-coded column groups—which might make compression also attractive for compute-intensive algorithm classes such as deep learning [57, 88]. For example, the NoScope system [47] selects difference and model filters to prune frames from video streams feeding into expensive convolutional neural networks. By representing frames as matrix rows, column-wise run-length encoding with column co-coding has a compression potential similar to such a difference detection.

Automatic Operator Fusion: Fused operators are increasingly used to avoid unnecessary materialized intermediates, to avoid unnecessary scans of input matrices, as well as to exploit sparsity across chains of operations. Spurred by a large development effort, recent work aims at the automatic generation of such fused operations [31]. So far, our CLA framework does not support operator fusion. However, as with query compilation for compressed data [55], operator fusion over compressed data is generally feasible and would significantly simplify the joint compiler integration of both compression and operation fusion. The challenges are the extension of existing templates for heterogeneous compression schemes, the cost model extension by compression statistics, and the fine-grained exploitation of sum-product optimization for techniques like pre-aggregation and post-scaling.

Special Value Handling: Furthermore, our CLA framework does not yet support special values such as NaN and ±INF, either in compressed matrices or uncompressed inputs to binary operations. Similar to sparse linear algebra operations, zero-suppressing offset lists ignore, for example, potential NaN outputs because $0 \cdot \text{NaN} = \text{NaN}$. The challenge is to efficiently support these special values without sacrificing performance—especially of sparse matrix operations—for the common case without NaNs. From an engineering perspective, all value-based operations require awareness of these special values because even simple assumptions such as self equivalence are violated ($\text{NaN} \neq \text{NaN}$).

Lossy Compression: Given the context of declarative ML algorithms, we aim at lossless matrix compression and exact linear algebra operations over these com-

pressed representations. However, existing systems like TensorFlow [2] and PStore [14] also apply lossy compression, specifically mantissa truncation, for network and disk I/O reduction, respectively. There are indeed good use cases for lossy compression and significant compression potential especially with regard to noise. The key challenge is to automatically identify opportunities where lossy compression is acceptable—such as the computation of gradients in early iterations—which bears similarities to approximate gradient decent via online aggregation [70]. Furthermore, there are also comprehensive algorithm-specific lossy compression techniques. For example, matrix factorization via random projections [38,85] identifies a relevant subspace—i.e., a subset of columns of the input matrix—via random sampling, maps the matrix into this subspace, and finally computes the factorization on the compressed representation.

7 Related Work

We generalize sparse matrix representations via compression and accordingly review related work of database compression, learning over compressed data, factorized learning, sparse linear algebra, graph compression, and compression planning.

Compressed Databases: The notion of compressing databases appears in the literature back in the early 1980s [9,26], 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 [66]. More recent systems that use bitmap-based compression include FastBit [92], Oracle [68], and Sybase IQ [83]. 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 [36], primarily with dictionary-based compression. Westmann et al. explored storage, query processing and optimization with regard to lightweight compression techniques [89]. Later, Raman and Swart investigated query processing over heavyweight Huffman coding schemes [71], 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 [81], SAP HANA [15,90], IBM DB2 with BLU Acceleration [72], Microsoft SQL Server [56], and HyPer [55]. Further, Kimura et al. made a case for compression-aware physical design tuning to overcome suboptimal design choices [51], which requires estimating sizes of compressed indexes. Existing estimators focus on compression schemes such as null suppression and dictionary encoding [43], 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 [13].

Learning over Compressed Data: SciDB [82]—as an array database—and the TileDB array storage manager [69] also use compression but both decompress arrays block-wise for each operation. Similarly, machine learning libraries on top of data-parallel frameworks such as MapReduce [29], Spark [95], or Flink [4] can leverage compression but need to decompress block-wise at the granularity of partitions. In contrast, there are algorithm-specific compressed data structures such as the CFP-tree and -array for frequent-itemset mining [79], as well as grammar-compressed matrices for partial least squares regression [84], which allow operations over the compressed representation. Very recently, Li et al. introduced tuple-oriented coding (TOC) [58], which extends the heavyweight LZW compression scheme for learning over compressed data, exemplified for Kmeans and GLM. This scheme uses a prefix tree as global dictionary while maintaining tuple boundaries and column indexes. Similar to our co-coding with pre-aggregation and post-scaling, partial results for distinct prefixes are computed only once, which reduces the number of floating point operations. In contrast, CLA focuses on the general case of linear algebra, heterogeneous encoding formats, and automatic compression planning.

Factorized Learning: Data originating from normalized relations requires join queries to construct the "flat", i.e., denormalized, input feature matrix. Denormalization can significantly increase the size and thus severely affect the performance of ML algorithms. Recent work on factorized learning [54,80], avoids this materialization by learning directly over normalized relations. Kumar et al. introduced factorized learning over primary-key/foreign-key joins—i.e., decomposing required computations and pushing them through joins—for GLM, Naïve Bayes and decision trees [53,54]. Further, Schleich et al. describes the learning of linear regression models over factorized joins for arbitrary join queries [65,80]. Similarly, Rendle discussed the learning of linear regression and factorization machines over a block structure, which is directly derived from the relational schema as part of feature engineering [74]. Beyond joins, other work manually overcomes unnecessary redundancy by preventing the subtraction of means from a large matrix which would turn it from sparse to dense [30]. For example, SystemML's algorithms for linear regression or GLM avoid shifting via special factor matrices that are then included into repeated gradient computations. In contrast to these algorithm-specific

approaches, TensorDB [50] pushes general tensor decompositions through joins and unions, and Morpheus [22] generalizes factorized learning by automatically rewriting linear algebra scripts into scripts over a logical data type for normalized data. In contrast to factorized learning, CLA requires us to compute such join queries, but immediately compresses the denormalized representations, potentially without materialization due to lazy evaluation in Spark. Overall, CLA and factorized learning share common principles but they are orthogonal. CLA applies to arbitrary datasets beyond the special case of denormalized data, while factorized learning avoids denormalization altogether.

Sparse Matrix Representations: Sparse matrix formats have been studied intensively in the literature. 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) [76]. These formats share the characteristic of encoding nonzero values along with their positions. Examples of hybrid formats—that try to combine advantages—are HYB (hybrid format) [10] that splits a matrix into ELL and COO areas to mitigate irregular structures, and SLACID [49] 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) [6] that apply 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 [91]. Finally, Kourtis et al. previously introduced compression techniques for sparse matrix formats, where they applied run-length encoding of column index deltas [48,52] and dictionary encoding [52]. In contrast to existing work, we aim at sparse and dense column value compression with heterogeneous encoding formats and column co-coding.

Graph Compression: A basic graph of nodes and edges is often represented either as a dense adjacency matrix, i.e., a squared boolean matrix indicating edges between nodes, or as sparse adjacency lists, i.e., a list of neighbors per node. In this context, various lossless and lossy compression schemes have been proposed in the literature [62]. In general, graph compression can be viewed as a special case of matrix compression that focuses on boolean matrices and specific operations such as reachability queries and graph pattern queries [33] or graph clustering and PageRank [8,77]. Interestingly, existing lossless schemes use (1) so-called compressor nodes [61], virtual nodes [20], or hyper nodes [33] to compress common edges to neighboring sub-graphs as well as (2) delta encoding of adjacency lists over reference nodes [3]. Both techniques are similar to our co-coding approach as they exploit correlation in terms of overlapping adjacency lists. However, we focus on the general case of compressing floating point matrices with co-coding and heterogeneous encoding schemes.

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 [25] is to experimentally compress a small segment of the data (excerpt) and observe the compression ratio. The drawbacks to this approach [40] 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 [39] 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 [40] 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, RLE, and DDC encoding formats 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 compressed size when several specific lightweight methods are applied to numeric matrices.

8 Conclusions

We have initiated work on value-based 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 as well as compression ratios similar or better compared to heavyweight formats like Gzip, and substantially 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. Note that we made the original version of CLA [32], as well as the extensions described in this paper, available open source in Apache SystemML 0.11 and SystemML 0.14, respectively. Finally, interesting future work includes (1) full optimizer integration, (2) global planning and physical design tuning, (3) alternative compression schemes, (4) operations beyond matrix-vector, and (5) automatic operator fusion.

Acknowledgments

We thank Alexandre Evfimievski and Prithviraj Sen for thoughtful discussions on CLA and code generation, Srinivasan Parthasarathy for pointing us to the related work on graph compression, as well as our reviewers for their valuable comments and suggestions.

References

1. D. J. Abadi et al. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, 2006.
2. M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*, 2016.
3. M. Adler and M. Mitzenmacher. Towards Compressing Web Graphs. In *DCC*, 2001.
4. A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB J.*, 23(6), 2014.
5. American Statistical Association (ASA). Airline on-time performance dataset. `http://stat-computing.org/dataexpo/2009/the-data.html`.
6. A. Ashari et al. An Efficient Two-Dimensional Blocking Strategy for Sparse Matrix-Vector Multiplication on GPUs. In *ICS*, 2014.
7. A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, 2015.
8. B. Bandyopadhyay et al. Topological Graph Sketching for Incremental and Scalable Analytics. In *CIKM*, 2016.
9. M. A. Bassiouni. Data Compression in Scientific and Statistical Databases. *TSE (Trans. SW Eng.)*, 11(10), 1985.
10. N. Bell and M. Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC*, 2009.
11. J. Bergstra et al. Theano: a CPU and GPU Math Expression Compiler. In *SciPy*, 2010.
12. K. S. Beyer et al. On Synopses for Distinct-Value Estimation Under Multiset Operations. In *SIGMOD*, 2007.
13. B. Bhattacharjee et al. Efficient Index Compression in DB2 LUW. *PVLDB*, 2(2), 2009.
14. S. Bhattacherjee et al. PStore: An Efficient Storage Framework for Managing Scientific Data. In *SSDBM*, 2014.
15. C. Binnig et al. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*, 2009.
16. M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.
17. M. Boehm et al. Declarative Machine Learning – A Classification of Basic Properties and Types. *CoRR*, 2016.
18. W. J. Bolosky and M. L. Scott. False Sharing and its Effect on Shared Memory Performance. In *SEDMS*, 1993.
19. L. Bottou. The infinite MNIST dataset. `http://leon.bottou.org/projects/infimnist`.
20. G. Buehrer and K. Chellapilla. A Scalable Pattern Mining Approach to Web Graph Compression with Communities. In *WSDM*, 2008.
21. M. Charikar et al. Towards Estimation Error Guarantees for Distinct Values. In *SIGMOD*, 2000.
22. L. Chen et al. Towards Linear Algebra over Normalized Data. *PVLDB*, 10(11), 2017.
23. R. Chitta et al. Approximate Kernel k-means: Solution to Large Scale Kernel Clustering. In *KDD*, 2011.
24. J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.
25. C. Constantinescu and M. Lu. Quick Estimation of Data Compression and De-duplication for Large Storage Systems. In *CCP*, 2011.
26. G. V. Cormack. Data Compression on a Database System. *Commun. ACM*, 28(12), 1985.
27. P. Damme et al. Lightweight Data Compression Algorithms: An Experimental Survey. In *EDBT*, 2017.
28. S. Das et al. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.
29. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
30. T. Elgamal et al. sPCA: Scalable Principal Component Analysis for Big Data on Distributed Platforms. In *SIGMOD*, 2015.
31. T. Elgamal et al. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. *CIDR*, 2017.
32. A. Elgohary et al. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12), 2016.
33. W. Fan et al. Query Preserving Graph Compression. In *SIGMOD*, 2012.
34. A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.
35. I. J. Good. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika*, 1953.
36. G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *Applied Computing*, 1991.
37. P. J. Haas and L. Stokes. Estimating the Number of Classes in a Finite Population. *J. Amer. Statist. Assoc.*, 93(444), 1998.
38. N. Halko et al. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2), 2011.
39. D. Harnik et al. Estimation of Deduplication Ratios in Large Data Sets. In *MSST*, 2012.
40. D. Harnik et al. To Zip or not to Zip: Effective Resource Usage for Real-Time Compression. In *FAST*, 2013.
41. B. Huang et al. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.
42. B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.
43. S. Idreos et al. Estimating the Compression Fraction of an Index using Sampling. In *ICDE*, 2010.
44. Intel. MKL: Math Kernel Library. `software.intel.com/en-us/intel-mkl/`.

45. D. S. Johnson et al. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM J. Comput.*, 3(4), 1974.
46. N. L. Johnson et al. *Univariate Discrete Distributions*. Wiley, New York, 2nd edition, 1992.
47. D. Kang et al. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB*, 10(11), 2017.
48. 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.
49. D. Kernert et al. SLACID - Sparse Linear Algebra in a Column-Oriented In-Memory Database System. In *SSDBM*, 2014.
50. M. Kim. *TensorDB and Tensor-Relational Model (TRM) for Efficient Tensor-Relational Operations*. PhD thesis, ASU, 2014.
51. H. Kimura et al. Compression Aware Physical Database Design. *PVLDB*, 4(10), 2011.
52. K. Kourtis et al. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *CF*, 2008.
53. A. Kumar et al. Demonstration of Santoku: Optimizing Machine Learning over Normalized Data. *PVLDB*, 8(12), 2015.
54. A. Kumar et al. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*, 2015.
55. H. Lang et al. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, 2016.
56. P. Larson et al. SQL Server Column Store Indexes. In *SIGMOD*, 2011.
57. Y. Lecun et al. Deep Learning. *Nature*, 521:436–444, 5 2015.
58. F. Li et al. When Lempel-Ziv-Welch Meets Machine Learning: A Case Study of Accelerating Machine Learning using Coding. *CoRR*, 2017.
59. M. Lichman. UCI Machine Learning Repository: Higgs, Covertype, US Census (1990). `archive.ics.uci.edu/ml/`.
60. S. Luo et al. Scalable Linear Algebra on a Relational Database System. In *ICDE*, 2017.
61. A. Maccioni and D. J. Abadi. Scalable Pattern Matching over Compressed Graphs via Dedensification. In *KDD*, 2016.
62. S. Maneth and F. Peternek. A Survey on Methods and Systems for Graph Compression. *CoRR*, 2015.
63. J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 1999.
64. NVIDIA. cuSPARSE: CUDA Sparse Matrix Library. `docs. nvidia.com/cuda/cusparse/`.
65. D. Olteanu and M. Schleich. F: Regression Models over Factorized Views. *PVLDB*, 9(13), 2016.
66. P. E. O'Neil. Model 204 Architecture and Performance. In *High Performance Transaction Systems*. 1989.
67. A. Or and J. Rosen. Unified Memory Management in Spark 1.6, 2015. SPARK-10000 design document.
68. Oracle. *Data Warehousing Guide, 11g Release 1*, 2007.
69. S. Papadopoulos et al. The TileDB Array Data Storage Manager. *PVLDB*, 10(4), 2016.
70. C. Qin and F. Rusu. Speculative Approximations for Terascale Analytics. *CoRR*, 2015.
71. V. Raman and G. Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*, 2006.
72. V. Raman et al. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 6(11), 2013.
73. S. Raskhodnikova et al. Strong Lower Bounds for Approximating Distribution Support Size and the Distinct Elements Problem. *SIAM J. Comput.*, 39(3), 2009.
74. S. Rendle. Scaling Factorization Machines to Relational Data. *PVLDB*, 6(5), 2013.
75. T. Rohrmann et al. Gilbert: Declarative Sparse Linear Algebra on Massively Parallel Dataflow Systems. In *BTW*, 2017.
76. Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2, 1994.
77. V. Satuluri et al. Local Graph Sparsification for Scalable Clustering. In *SIGMOD*, 2011.
78. S. Schelter et al. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. *NIPS Workshop MLSystems*, 2016.
79. B. Schlegel et al. Memory-Efficient Frequent-Itemset Mining. In *EDBT*, 2011.
80. M. Schleich et al. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*, 2016.
81. M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.
82. M. Stonebraker et al. The Architecture of SciDB. In *SSDBM*, 2011.
83. Sysbase. *IQ 15.4 System Administration Guide*, 2013.
84. Y. Tabei et al. Scalable Partial Least Squares Regression on Grammar-Compressed Data Matrices. In *KDD*, 2016.
85. M. Tepper and G. Sapiro. Compressed Nonnegative Matrix Factorization Is Fast and Accurate. *IEEE Trans. Signal Processing*, 64(9), 2016.
86. Y. Tian et al. Scalable and Numerically Stable Descriptive Statistics in SystemML. In *ICDE*, 2012.
87. 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.
88. W. Wang et al. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Record*, 45(2), 2016.
89. T. Westmann et al. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3), 2000.
90. T. Willhalm et al. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1), 2009.
91. S. Williams et al. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In *SC*, 2007.
92. K. Wu et al. Optimizing Bitmap Indices With Efficient Compression. *TODS*, 31(1), 2006.
93. L. Yu et al. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, 2015.
94. R. B. Zadeh et al. Matrix Computations and Optimization in Apache Spark. In *KDD*, 2016.
95. M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.
96. C. Zhang et al. Materialization Optimizations for Feature Selection Workloads. In *SIGMOD*, 2014.
97. M. Zukowski et al. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.