AWARE: Workload-aware, Redundancy-exploiting Linear Algebra

SEBASTIAN BAUNSGAARD*, Technische Universität Berlin, Germany
MATTHIAS BOEHM*, Technische Universität Berlin, Germany

Compression is an effective technique for fitting data in available memory, reducing I/O, and increasing instruction parallelism. While data systems primarily rely on lossless compression, modern machine learning (ML) systems exploit the approximate nature of ML and mostly use lossy compression via low-precision floating- or fixed-point representations. The resulting unknown impact on learning progress, and model accuracy, however, create trust concerns, that require trial and error, and are problematic for declarative ML pipelines. Given the trend towards increasingly complex, composite ML pipelines—with outer loops for hyper-parameter tuning, feature selection, and data cleaning/augmentation—it is hard for a user to infer the impact of lossy compression. Sparsity exploitation is a common lossless scheme used to improve performance without this uncertainty. Evolving this concept to general redundancy-exploiting compression is a natural next step. Existing work on lossless compression and compressed linear algebra (CLA) enable such exploitation to a degree, but face challenges for general applicability. In this paper, we address these limitations with a workload-aware compression framework, comprising a broad spectrum of new compression schemes and kernels. Instead of a data-centric approach that optimizes compression ratios, our workload-aware compression summarizes the workload of an ML pipeline, and optimizes the compression and execution plan to minimize execution time. On various micro benchmarks and end-to-end ML pipelines, we observe improvements for individual operations up to 10,000x and ML algorithms up to 6.6x compared to uncompressed operations.

CCS Concepts: • **Theory of computation** → **Data compression**; • **Information systems** → **Data compression**; *Data scans*; • **Computing methodologies** → *Linear algebra algorithms*; *Machine learning*.

Additional Key Words and Phrases: Redundancy Exploitation; Lossless Compression; Linear Algebra; Machine Learning; Workload-aware Optimization; Large-scale; Declarative; Online Compression

ACM Reference Format:
Sebastian Baunsgaard and Matthias Boehm. 2023. AWARE: Workload-aware, Redundancy-exploiting Linear Algebra. *Proc. ACM Manag. Data* 1, 1, Article 2 (May 2023), 28 pages. https://doi.org/10.1145/3588682

1 INTRODUCTION

Data compression dates back to old Morse code (from the eighteen-hundreds) that uses shorter codewords for common letters [80]. In modern data management and machine learning (ML) systems, compression is a well-established and effective technique for fitting data in available memory, reducing I/O and memory bandwidth requirements [25, 62], and increasing instruction parallelism [79]. Data management systems with declarative interfaces almost exclusively rely on lossless compression in order to ensure correct results, and lightweight techniques [21, 22] that offer a good balance of compression ratios and (de)compression speed.

*This work was partially done at Graz University of Technology, Austria.

Authors' addresses: Sebastian Baunsgaard, Technische Universität Berlin, Berlin, Germany; Matthias Boehm, Technische Universität Berlin, Berlin, Germany.

2836-6573/2023/5-ART2 $15.00
https://doi.org/10.1145/3588682

Compression: In contrast to lossless compression in data systems, ML systems—especially for mini-batch training of deep neural networks (DNN) predominately exploit the approximate nature of ML models and apply lossy compression such as quantization (i.e., static or dynamic discretization) [30, 84], sparsification (clipping of low quantities) [29, 56], new data types (e.g., `bfloat16`, `TF32`) [39, 56, 65], dimensionality reduction [34] and sampling (few step/epoch mini-batch training [72], or sampled batch training [60]). However, lossy compression introduces unknown behavior on new datasets and models, which creates trust concerns and requires an exploratory trial-and-error process [78]. In contrast, lossless compression ensures result correctness, but is less commonly applied in ML. Examples include—besides general-purpose lossless matrix compression like Snappy or LZ4—value-indexed representations [37, 40], grammar-compressed matrices [73], tuple-oriented coding (TOC) [45] and Compressed Linear Algebra (CLA) [25, 26].

Redundancy Exploitation: Sparsity exploitation is currently a major trend across the stack from hardware [56, 58], over systems [14, 50, 70], to algorithms [28, 29, 63], but its applicability is limited to sparse data (many zero values). Previous work on compressed linear algebra (CLA) [25, 26] further allowed for more general redundancy exploitation (with repeated values and correlation) by applying lightweight lossless compression techniques like dictionary, run-length, and offset-list encoding and executing linear algebra operations like matrix-vector multiplications and element-wise operations directly on compressed representations. CLA was integrated into Apache SystemML [13], but by default only applied for multi-column matrices, whose size exceed aggregated cluster memory, and all operations are supported in compressed space. These constraints ensure that online compression overheads are amortized but limit applicability in practice.

***Aware* Goals and Contributions:** We aim to improve the applicability of lossless matrix compression in complex ML pipelines. The key objective is to optimize for execution time of a given workload instead of compression ratios. This metric also covers reduced compression time to amortize online compression, optimization for size if data access is the bottleneck, and fast operations via specialized compression decisions, kernels, and execution plans. To this end, we introduce a workload-aware matrix compression framework (for full matrices or tiles of a distributed matrix), and make the following detailed technical contributions:

- *Compression Framework:* New encodings and compressed operations (Section 3 and 4), which are designed for compressed intermediates and thus, chains of operations.
- *Workload-aware Compression:* Novel workload-aware compression planning and compilation techniques (Section 5).
- *Experiments:* Local & distributed experiments comparing uncompressed linear algebra (ULA), CLA [25, 26], TensorFlow, and *Aware* on various workloads (Section 6).

2 BACKGROUND AND OVERVIEW

This section reviews the main characteristics of CLA and its limitations, which directly motivate key design decisions of *Aware*. CLA [25, 26] is a lossless compression framework leveraging column-wise compression, column co-coding (encode groups of columns as single units), and heterogeneous column encoding schemes. This design exploits characteristics of feature matrices—with categorical and numerical features in columns—namely, tall and skinny matrices, non-uniform sparsity, low cardinality and correlations. Since the selection of column groups is strongly data-dependent, CLA introduced a sampling-based compression planning for online compression after an initial read of an input matrix. Once compressed, specialized kernels work directly on the compressed representation if applicable and efficient, otherwise fall back to decompression followed by uncompressed kernels.

Table 1. Differences of CLA and *Aware*

	CLA [25, 26]	***Aware***
Co-Coding	$O(m^2)$	$O(m)$
Column Group Encodings	4 (5)	7 (327)
Materialization	Eager	Deferred
Optimization Objective	Data	Data & Ops
Matrix Multiplication	MV, VM	MV, VM, MM

2.1 Limitations of CLA

Despite CLA's compelling properties—allowing operations directly on the compressed representations—there are limitations and missing functionality that hinder general applicability.

Compression Costs: The original CLA's [25] sampling-based compression planning used a sample fraction and a co-coding algorithm that—ignoring greedy partitioning—required $O(m^3)$ group extractions from the sample. The refined CLA [26] improved the co-coding approach via memoization to $O(m^2)$ group extractions. However, despite column classification into compressible and incompressible columns, matrices with no or minor compression benefits were recognized much too late, after incurring already substantial overhead. Furthermore, the super-linear co-coding complexity becomes infeasible for millions of columns.

Compressed Intermediates: CLA performs online compression for input matrices and keeps outputs of amenable operations like matrix-scalar element-wise multiplications, or scaling and shifting compressed. Other operations produce uncompressed outputs (e.g., after feature transformations or data cleaning). In complex, exploratory ML pipelines there are multiple sources of redundancy though, which would largely benefit from a more fine-grained selection of intermediates and optimization of their individual compression schemes.

Optimization Objective: The design and implementation of CLA aimed at the sweet spot of compressed operation performance close to uncompressed (low in-memory overhead), while achieving good compression ratios (fit larger data in memory, reduced I/O for large data). Besides this hand-crafted tuning, the internal objective for selecting compression schemes and co-coding then only focuses on minimizing compressed size. Fundamentally, however, the overall optimization objective should be total execution time, factoring in compression, compressed operations, and potential I/O in order to better adapt to data, operation, and cluster characteristics.

2.2 *Aware* Overview

Our workload-aware compression addresses these limitations with new techniques for compression planning, compressed intermediates, and different optimization objectives. Table 1 highlights these key differences between CLA and *Aware*. First, to reduce compression time, we introduce a new co-coding technique that performs group *combinations* instead of group *extractions*, reducing the overhead to analyze groups. Our co-coding approach further includes a new enumeration heuristic that only evaluates $O(m)$ group combinations. Second, for extended utilization of compressed intermediates, we provide new column group encodings that facilitate shallow-copy operations. Table 1 shows the number of high-level column group types and—in parenthesis—the total number of variations of these encodings. We also introduce a deferred operation/encoding design, where compressed operations can output different types of encodings, allowing compressed intermediates where CLA would decompress or return inefficient representations. Furthermore, *Aware* natively supports compressed matrix-matrix multiplication (even with two compressed inputs), unlike CLA that would process it via repeated matrix-vector multiplications and thus decompresses one

Table 2. Overview of CLA and *Aware* Column Groups.

Type	Description	CLA [25, 26]	*Aware*
CON	Constant or Empty Columns		✓
DDC	Dense Dictionary Coding	✓	✓
OLE	Offset-list Encoding	✓	(✓)
FOR	Frame of Reference		✓
RLE	Run-length Encoding	✓	(✓)
SDC	Sparse Dictionary Coding		✓
UC	Uncompressed (dense/sparse)	✓	✓

side. And above all, *Aware* uses a cost-based optimization objective of minimizing the workload execution time and thus, tuning the compression process, the compressed representation, and compressed operations in a principled way.

3 COMPRESSION

This section describes *Aware*'s compressed representation, selected new concepts, and the overall compression algorithm. The new encoding schemes are designed for redundancy exploitation across operations, while the new compression algorithm ensures fast, easy to amortize compression.

3.1 Compressed Representation

Aware encodes each column group independently in a specific encoding type. Table 2 shows these column-group encodings, as well as the differences to CLA. Figure 1 then presents an example of compressing a 10×6 matrix into three single-column groups (0, 2 and 5), one two-column group ($\{1, 3\}$), and an empty group. Also shown is an element-wise scalar subtraction on the compressed matrix, creating two alternative compressed outputs (A and B).

Dense Dictionary Coding (DDC) contains two parts: a *dictionary* with the distinct value tuples in the column group, (shown in Figure 1 as a Dict with 2 values for column $\{0\}$), and an *index structure* with a row-to-tuple mapping (e.g., dictionary position). DDC is dense because each row input is assigned a code in the map.

Sparse Dictionary Coding (SDC) is a combination of DDC and sparse matrix formats like compressed sparse rows (CSR). An example is shown in yellow for columns $\{1, 3\}$ in Figure 1. Like DDC, each group has a dictionary of all unique tuples except the most frequent tuple named "Def" for default. This scheme encodes row locations of non-default tuples in the index structure as row-index pairs. This approach is similar to compressed sparse columns (CSC) that store row-index/value pairs for non-zero values, but extends it for general redundancy exploitation (default values, dictionary references). The row part is further specialized to delta offsets ("Off") from previous rows to allow smaller physical codewords. Finally, a "Map" (index part of CSC) maps offsets to tuples in the dictionary, similar to DDC. SDC specializes into SDCZero, where zero default entries are removed like in the $\{2\}$ blue column group, and SDCSingle for binary data (one dictionary entry, one default), removing the need of codes like in the $\{4\}$ orange column group.

Frame of Reference (FOR) is used as a second layer on top of DDC or SDC (called DDCFOR, SDCFOR). This encoding shallow-copies the index structures and dictionaries, and allocates a reference tuple, that indicates a global value offset. An SDC group can zero-out the default tuple by adding it to the dictionary and subtracting it from the reference tuple (converted to SDCZero).

Offset-list Encoding (OLE) is a CLA encoding scheme, but largely superseded by SDC. SDC has in general worse compression than OLE, but SDC allow the group to densify its values without

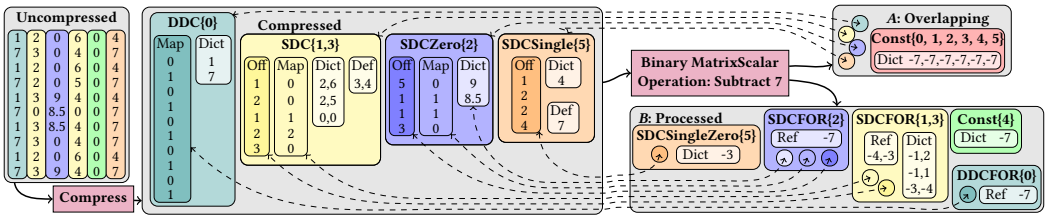


Fig. 1. Example *Aware* Matrix Compression and Operation (with Alternative Compressed Outputs).

having to modify its index structure, while if densified OLE encode a value for each row making it inefficient and potentially bigger.

Run-length Encoding (RLE) is unlikely beneficial in *Aware* since co-coding many columns—which is good for operation performance, and likely to be done in scenarios with good RLE compression—makes it unlikely to retain sufficiently long runs. RLE also reallocates the index structure on densifying operations.

Constant Encodings are used for empty, constant columns, and constant tuple column groups. CLA encodes such groups using run-length encoding (RLE). Instead we specialize with constant groups in order to simplify operations with compressed outputs.

Dictionaries: CLA uses basic `FP64` dictionaries. In contrast, *Aware* generalizes the data binding of dictionaries and uses basic `FP64` and `INT8` arrays, or sparse matrices. The more columns co-coded, the more zeros might be included in unique tuples and thus, warrant a sparse dictionary. *Aware* does not share dictionaries across multiple column groups like CLA does in some cases.

Index Encodings: The different column group implementations share common primitives such as Map and Off, of different value types (not shown in the figure). Map supports encodings in Bit, Byte, UByte, Char, Char+Byte and Int, while Off supports delta-encoded Byte or Char arrays, and specializations for one/two offsets.

Overlapping Column Groups: *Aware* allows column groups to overlap with partial sum semantics. Multiple column groups may refer to the same column but store separate dictionaries and index structures. Overlapping helps column groups preserve (and due to compression, eliminate) structural redundancy of intermediates for chains of operations such as matrix multiplication, row sums aggregation, and scalar or column addition.

An Operation Example: Figure 1 (right) shows an operation example subtracting 7 from the compressed matrix. Option *A* creates an overlapping representation with pointers to the input column groups and a new constant group subtracting 7 from the entire matrix. In contrast, Option *B* performs the subtraction on all column groups, creating different output group types. The empty column becomes a *Const* group of -7. Column {2} in blue becomes a *SDCFOR* group that copies pointers to the previous dictionary and index structure, and only materializes a new reference value. The *SDCSingle* group in orange becomes an *SDCSingleZero* because Def 7-7 yields 0. The *SDC* group in yellow has a different default value, and thus, produces an *SDCFOR* group, where we subtract the default value from the dictionary, and subtract 7 from the default value as new reference value. The total costs of Option *A* is 1 FLOP and allocation of small arrays and pointers. Option *B* requires 13 FLOPs but outputs a non-overlapping state, which can be beneficial for following operations. Uncompressed requires 60 FLOPs and an allocation in the input size. In contrast to *Aware*, CLA with DDC{0} DDC {1, 3}, DDC {5}, OLE {2} and RLE {5} compression requires 16 FLOPs (3 more due to OLE/RLE/DDC), but more significantly, allocates new index structures in columns {2} and {5}. Our current heuristic for such additive scalar operations is to return an overlapped representation (with a new/reused constant group) if the input was overlapping, and processed groups otherwise.

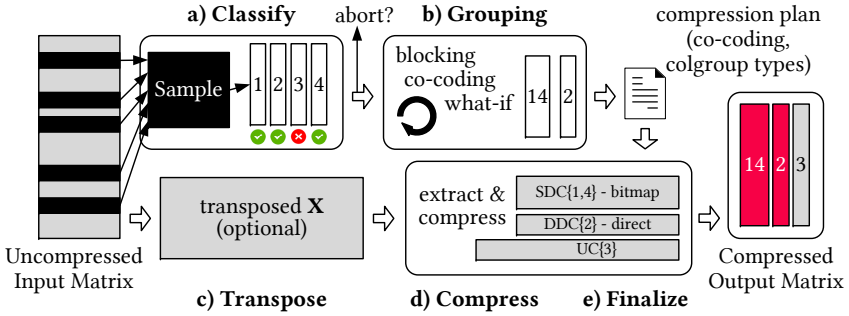


Fig. 2. Workflow of Compression Phases.

Algorithm 1 Compression Algorithm

Require: Matrix input M
  return M_c
  G ← ExtractIndexStructures(Sample(M))
  SingleColumInfos ← Classify(G) ▷ Abort 1
  Plan ← Grouping(SingleColumInfos, G) ▷ Abort 2
  (M, t) ← TransposeMaybe(Plan, M) ▷ t is true if M is transposed
  M_c ← Finalize(Compress(Plan, M, t)) ▷ Abort 3

3.2 Compression Algorithm

Our compression algorithm aims to reduce the online compression[1] time, introduce workload-awareness via generic cost functions (computation, memory or combinations), and handle matrices with many columns. Together, solutions to these issues, allow us to apply compression for a wide variety of inputs and intermediates with robust performance improvements. Given an uncompressed matrix, the *Aware* compression algorithm (Figure 2 and Alg 1) comprises the following phases:

a) Classify: For efficient compression planning, we first obtain an index structure (dense or sparse for DDC or SDC) for each column in a sample of the input matrix, as well as counts of non zeros (NNZ) per column in the input matrix. Using the index structure and NNZ count, we compute summary statistics for individual columns (e.g., the frequency of distinct items), estimate the cost of the individual columns, classify columns as compressible or incompressible, and extract empty columns. For classifying a column or list of columns, the same summary statistics are needed, irrespective of optimizing for workload cost or size in memory. Compared to the CLA compression algorithm—where the entire uncompressed matrix was transposed first for efficient extraction in Classify and Compress—we benefit from working only with small index structures until deciding on aborting the compression for non-amenable matrices. Furthermore, we gain more efficient sample extraction, and bounded temporary memory requirements for incompressible matrices.

b) Grouping: Column co-coding seeks to find column groups in order to exploit redundancy among correlated columns. *Aware* introduces two techniques to improve CLA's co-coding algorithm. First, instead of extracting statistics from the sample when combining columns, we combine the index structures of two already extracted groups from the classification phase or previously combined columns. Algorithm 2 combines two dense index structures (I^r and I^l) into a combined index structure I^c. This algorithm allocates a mapping M that is able to encode all possible unique mappings from combining I^r and I^l by the product of their numbers of distinct items d_l and d_r.

[1]Online compression refers to the compression of inputs or intermediates during runtime of a linear algebra program (e.g., after reading uncompressed inputs).

Algorithm 2 Combine Algorithm for Dense Index Structures

Require: Index structures for two groups I^l, I^r
  return Combined index structure I^c
  $M \leftarrow I[d_l \cdot d_r]$, $u \leftarrow 1$ ▷ Allocate map of possible distinct size
  for $i \leftarrow 0$ to n **do**
    $m \leftarrow I^l_i + I^r_i \cdot d_r$ ▷ Calculate new unique index
    if $M_m = 0$ **then** ▷ Non-existing value at the unique index
      $M_m \leftarrow u$++ ▷ Assign unique index to next unique value
    $I^c_i \leftarrow M_m - 1$ ▷ Assign output to map value at unique index

Algorithm 3 PriorityQueue Co-coding Algorithm

Require: A queue of all current index structures Q
  return A list of index structures G
  while $Q.peek \neq NULL$, $I^l \leftarrow Q.poll$ **do** ▷ Remove cheapest Index
    $I^r \leftarrow Q.peek$ ▷ Look at next cheapest Index
    $I^c \leftarrow combine(I^l, I^r)$ ▷ Combine two cheapest
    if $I^c.cost < I^l.cost + I^r.cost$ **then** ▷ Costs of combined is lower
      $Q.poll$, $Q.put(I^c)$ ▷ Remove I^r from queue and add I^c
    else
      $G.add(I^l)$ ▷ Add cheapest (already extracted) to output

Further specializations are algorithms for sparse-sparse and sparse-dense combining. Second, we introduce a new co-coding algorithm (see Algorithm 3) that uses a priority queue Q for sorting columns (or column groups) based on a configurable cost function, and combines groups at the head of the queue. We found that starting with this new co-coding algorithm and switching to a greedy combining approach at a threshold number of remaining groups gives a good balance of compression time and quality. In cases with millions of columns, we do a static partitioning of the columns to available threads and combine columns in a thread-local manner.

c) Transpose: The uncompressed input matrix can be transposed (columns in row-major) if the compress phase would benefit from sequential access and amortize such data reorganization. This decision is dependent on the data characteristics (e.g., matrix dimensions, dense or sparse) and the chosen compression plan (e.g., co-coded columns).

d) Compress: During compression, we take the input matrix and compression plan (co-coding decisions, and column-group types), and create the compressed column groups. For every group, we first extract its single- or multi-column uncompressed bitmap as a canonical representation of distinct tuples and offset lists per tuple. With these temporary offset lists, we re-evaluate the group types, and finally create the physical encoding of the compressed column groups, which involves various specializations (e.g., delta-encoded offsets) for smaller code words. Once a column group is compressed—and it is beneficial in terms of workload costs—we analyze if we can sparsify its dictionary via a frame-of-reference encoding, and if so apply the transformation. In contrast to CLA, we apply no corrections for estimated compressible but actually incompressible columns because the estimators and co-coding show robust behavior.

e) Finalize: In a last phase, we perform compaction of special groups, and compare costs of the actual compressed representation with the uncompressed costs (and abort if needed). Finally, we cleanup all temporary buffers but keep a soft reference (subject to garbage collection under memory pressure) to the uncompressed block to skip potential decompressions.

Parallelization Strategies: When compressing distributed matrices, blocks are compressed independently in a data-parallel manner with single-threaded compression per block. In contrast, local, in-memory compression utilizes multi-threading with barriers per phase. Classify parallelizes over columns, Grouping over blocks of columns, Compress over column groups and in some cases row partitions, and Transpose uses a multi-threaded cache-conscious uncompressed transpose operation. A more fine-grained parallelization with a task graph [53] is interesting future work.

4 COMPRESSED OPERATIONS

Performing linear algebra operations—like matrix multiplications, element-wise operations, and aggregations—on compressed matrices can improve memory-bandwidth requirements and cache utilization, reduce the number of floating point operations, and preserve structural redundancy across chains of operations. *Aware* makes extensions for compressed matrix-matrix multiplications and compressed intermediates, which broaden its applicability.

4.1 Design Principles and Notation

As a basis for discussing compressed operations, we first establish necessary terminology, and summarize underlying design principles.

Definitions and Scope: We define *sparse-safe* operations as operations or aggregations that only need to process non-zero input cells. For example, round($\mathbf{X}$) is sparse-safe, while exp($\mathbf{X}$) is sparse-unsafe because $\exp(0) = 1$. *Special values* like NaN (not-a-number, with $\text{NaN} \cdot 0 = \text{NaN}$) are not supported compressed because they render sparse linear algebra invalid [70].

Design Principles: Many of the *Aware* operations share the following design principles. Compared to CLA, *Aware* applies these principles to more operations and generalizes them with the goals of redundancy exploitation and minimizing total execution time.

- *Shared Index Structures:* For operations that only modify distinct values (e.g., $\mathbf{X} \cdot 7$), we perform dictionary-local operations and shallow-copy the index structures into the output.
- *Memoized Tuple Frequencies:* Operations like sum($\mathbf{X}$) aggregate the distinct tuples scaled by their frequencies. To avoid redundant computation, we memoize computed frequencies and retain them on shallow-copies of indexes.
- *Exploited Structural Redundancy:* While many sparse-unsafe operations can be executed on compressed matrices, they can require the materialization of zero, which often creates large unbalanced groups. Instead, in *Aware* we exploit both sparsity and redundancy via the handling of default values, as well as preserve structural redundancy across operations.
- *Soft References:* We keep useful but recomputable data structures (e.g., decompressed data, offset pointers to indexes, and tuple frequencies) on soft references. Any serialization or memory estimates do not include these cached objects.

Notation: Finally, we need some additional notation. An $n \times m$ uncompressed input matrix is compressed into a set of column groups $\mathcal{G}$, where $|\mathcal{G}|$ denotes the number of column groups (with $|\mathcal{G}| \leq m$ without overlap), and $\mathcal{G}_i$ denotes the i-th column group. A single column group $\mathcal{G}_i$ comprises $|\mathcal{G}_i|$ columns, a $d_i \times |\mathcal{G}_i|$ dictionary $\mathbf{D}_i$ with d_i distinct tuples, and an index structure $\mathbf{I}_i$. For matrix multiplications $\mathbf{A}\,\mathcal{G}$ or $\mathcal{G}\,\mathbf{B}$, let k denote the number of rows in $\mathbf{A}$ and columns in $\mathbf{B}$, respectively. Given a matrix or vector $\mathbf{X}$, $nnz(\mathbf{X})$ denotes its number of non-zeros and $nnd(\mathbf{X})$ denotes its number of non-default values (equivalent to $nnz(\mathbf{X})$ if zero is the most frequent value).

4.2 Matrix Multiplications

CLA supports only matrix-vector and vector-matrix multiplications directly on compressed representations, but emulates matrix-matrix multiplications via repeated slicing and matrix-vector

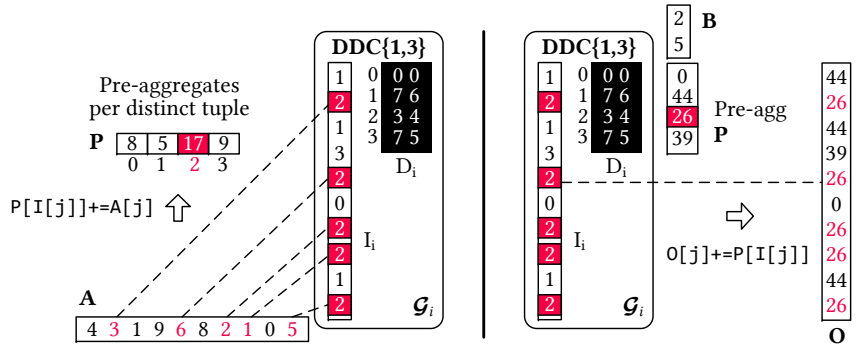


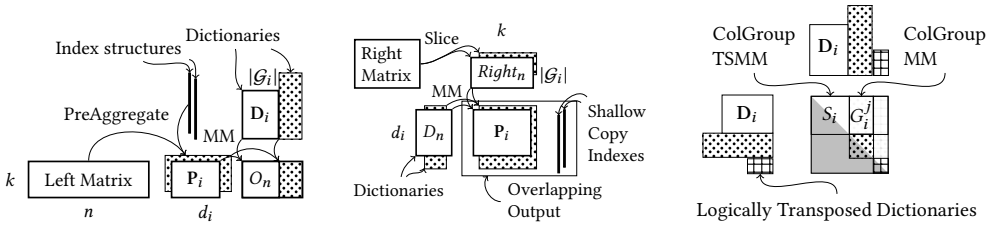
Fig. 3. Example Pre-aggregation in Compressed MatMult.

multiplications. This approach provides simplicity and reasonable performance, but looses performance as the size of the second matrix increases, which is common in applications like multi-class classification, dimensionality reduction, and clustering. Other previous work like TOC [45] supports matrix-matrix multiplication but belongs to the class of grammar-compression. In this section, we introduce simple yet impactful techniques for matrix-matrix multiplications on lightweight compressed matrices, including special cases of compressed-compressed multiplication.

Preaggregation: A central technique of compressed matrix multiplications are different forms of pre-aggregation over the distinct tuples. In *Aware*, we vectorize such pre-aggregation for improved simplicity and performance. For instance, Figure 3 shows the intuition of pre-aggregation in left and right matrix multiplication on our running example. First, for a left matrix multiplication $\mathbf{A}\,\mathcal{G}$ with an uncompressed vector $\mathbf{A}$, we initialize a zero vector $\mathbf{P}$, and accrue the entries of A according to indexes $\mathbf{I}_i$. Subsequently, a simple uncompressed vector-matrix multiplication $\mathbf{P}\,\mathbf{D}_i$ of the pre-aggregates and the dictionary yields the overall result (for columns of the column group). Instead of multiplying all entries with the same distinct value (or tuple in case of co-coding), we simply distribute multiplication over addition. Second, for a right matrix multiplication $\mathcal{G}\,\mathbf{B}$ with an uncompressed vector $\mathbf{B}$ (subset relevant to the column group), we first compute a matrix-vector multiplication of $\mathbf{D}_i\,\mathbf{B}$ to get the pre-aggregated vector $\mathbf{P}$, and subsequently add these pre-aggregated values to the output according to indexes $\mathbf{I}_i$. Interestingly, a similar pre-aggregation strategy is also applied as a general case for unnesting correlated subqueries [54]. Given this vectorized form and the storage of dictionaries as uncompressed matrices, we can directly apply cache-conscious uncompressed matrix multiplications for the general case of left- or right-hand-side uncompressed matrices with k rows or columns, respectively.

Left Matrix Multiplication: We call the matrix multiplication ($\mathbf{A}\,\mathcal{G}$)—where the left-hand-side input $\mathbf{A}$ is uncompressed—a left matrix multiplication (LMM). Figure 4(a) shows an LMM with two column groups. For each column group, we compute the pre-aggregated $k \times d_i$ matrix $\mathbf{P}_i$ (via the already described vectorized pre-aggregation), then matrix multiply $\mathbf{P}_i\,\mathbf{D}_i$, and finally, shift these results into the correct column positions of the output matrix $\mathbf{O}$. Pre-aggregation for each column group is a linear scan of $\mathbf{A}$, but for large index structures $\mathbf{I}_i$, we can utilize cache-blocking to reuse blocks of $\mathbf{I}_i$ from caches across multiple rows in $\mathbf{A}$. The more co-coding is applied, and/or the smaller the number of distinct items per group, the more we benefit from LMM pre-aggregation in terms of reduced floating point operations and data accesses. Multi-threaded LMM operations parallelize over column groups and rows in $\mathbf{A}$ because they access disjoint output columns.

Morphing: Some types of column groups are able to skip processing rows or columns for certain operations. We leverage such properties by changing the format—similar to the technique from MorphStore [23]—of DDCFOR, SDC, SDCFOR, and SDCSingle groups before performing LMM.



(a) Left Matrix Multiply ($\mathbf{A}\,\mathcal{G}$). (b) Right Matrix Multiply ($\mathcal{G}\,\mathbf{B}$) (c) Transpose-Self MM($\mathcal{G}^\top\,\mathcal{G}$)

Fig. 4. Types of Compressed Matrix Multiplication (with 2 and 3 column groups).

Using SDCFOR as an example, we simply take the reference tuple as a vector, subtract it from the dictionary, and return a SDCZero column group with the modified dictionary. We multiply **A** and the column group using the preaggregate technique, and then add the vector scaled by row-sum of the left-hand-side matrix. In cases where multiple groups are morphed, their vectors are combined and processed together. This technique is also applied—with slight modifications—in decompression, right matrix multiply, and unary aggregates. This approach is virtually constructing an overlapping constant group for the entire matrix, transforming a single multiplication into a cheaper matrix multiplication, a row sum, and vector outer-product.

Right Matrix Multiplication: Similar to LMM, we call a matrix multiplication ($\mathcal{G}\,\mathbf{B}$)—where the right-hand-side input **B** is uncompressed—a right matrix multiplication (RMM). Figure 4(b) shows an example with two column groups. Our column-oriented compression and multiplication by **B** from the right, provides an opportunity to preserve structural redundancy and thus, avoid unnecessary decompression (aggregation into an uncompressed output). The simple, yet very effective, key idea of our RMM is to only perform the vectorized pre-aggregation $\mathbf{P}_i = \mathbf{D}_i\,\mathbf{B}^*$ by multiplying the column group dictionaries with related rows in **B**, and then store these pre-aggregates as new dictionaries of overlapping column groups. This way, we can leave the index structures $\mathbf{I}_i$ untouched and shallow-copy them into the compressed output representation, preserving the source redundancy. Each output column group now has dictionaries of size $d_i \times k$. The individual column groups compute, again independent (but now overlapping) outputs and thus, multi-threaded operations parallelize over column groups and columns of **B**. In distributed environments with block-local matrix multiplications, the same RMM applies and the overlapping output can be preserved (if beneficial in size) even through serialization and shuffling.

Transpose-Self Matrix Multiplication: Transpose-self matrix multiplication (TSMM[2]) $\mathbf{X}^\top\,\mathbf{X}$ or $\mathbf{X}\,\mathbf{X}^\top$—whose outputs is known to be symmetric—appears in many applications such as closed-form linear regression, co-variance and correlation matrices, PCA, and distance computations. CLA emulates TSMM again via slicing and repeated vector-matrix multiplications. In contrast, *Aware* natively supports TSMM as shown in Figure 4(c), as well as compressed-compressed matrix multiplications (with transposed left input), which are also commonly occurring in practice. A TSMM is composed of pairs of column group operations, where blocks on the diagonal are self-joins of column groups, while others are $|\mathcal{G}| \cdot (|\mathcal{G}| - 1)/2$ combinations of column groups. First, a self-join of a column group computes (or reuses) the $d_i \times 1$ pre-aggregate **P** of tuple frequencies, and subsequently computes the results via an uncompressed TSMM $(\mathbf{D} \odot \mathbf{P})^\top\,(\mathbf{D}i)$, i.e., where rows of **D** are scaled by their frequencies. Second, for remaining blocks and compressed-compressed matrix multiplications, we adopt the strategy of LMM: pre-aggregation and matrix multiplication $\mathbf{P}_i\,\mathbf{D}_i$. However, given two compressed inputs, we can freely pick the pre-aggregation side and

[2]TSMM is also known as BLAS syrk (symmetric rank-k update) or Gram matrix.

Table 3. Complexity of Matrix Multiplications.

	Uncompressed (dense)	***Aware*** (multiple, dense column groups)
LMM	$O(knm)$	$O\left(kn\lvert\mathcal{G}\rvert + k\sum_{i=1}^{\lvert\mathcal{G}\rvert}(d_i\lvert\mathcal{G}_i\rvert)\right)$
RMM	$O(knm)$	$O\left(k\sum_{i=1}^{\lvert\mathcal{G}\rvert}(d_i\lvert\mathcal{G}_i\rvert)\right)$
TSMM	$O\left(nm^2\right)$	$O\left(l\sum_{i=1}^{\lvert\mathcal{G}\rvert}(n + d_i\lvert\mathcal{G}_i\rvert^2) + \sum_{i=1}^{\lvert\mathcal{G}\rvert}\sum_{j=i+1}^{\lvert\mathcal{G}\rvert}(n\lvert\mathcal{G}_i\rvert + \lvert\mathcal{G}_i\rvert d_j\lvert\mathcal{G}_j\rvert)\right)$

alternatively, do $(\mathbf{P}_j\,\mathbf{D}_j)^\top$. In any case, the pre-aggregate is computed without decompression and can exploit column-group characteristics (e.g., sparse, non-default, or constant encoding).

Cost Analysis: Apart from reduced I/O and memory-bandwidth requirements due to the smaller compressed size, compression can also reduce the number of floating point operations. In detail, Table 3 compares the asymptotic behavior of uncompressed matrix multiplications with related *Aware* operations. Uncompressed LMM and RMM have both cubic complexity, where we ignore sparse linear algebra for the sake of simplicity. In contrast, compressed LMM and RMM have a complexity has depends on the data characteristics (distinct items and co-coding per column group). First, for LMM, we have two terms for pre-aggregation (only additions) and dictionary-based computation (additions and multiplications). With substantial co-coding (e.g., $|\mathcal{G}| = 1 \wedge |\mathcal{G}_i| = m$) and few distinct items d_i, a much better complexity than cubic is possible ($O(kn + km)$ instead of $O(knm)$). In the worst-case, $|\mathcal{G}| = m$ and $d_i = n$, which gives $O(knm)$ but with a higher constant factor. Second, RMM has only the second term of LMM and thus, we already benefit with less favorable data characteristics but the same worst-case guarantees apply, even with subsequent decompression. TSMM is more complex but the first term of the addition represents self-joins per column group (including pre-aggregation), and the second term enumerates pairs of column groups with two sub-terms for pre-aggregation and scaling. These cost functions, together with estimated or observed compression ratios, are also the basis for our workload-aware compression planning, allowing us to optimize for total execution time.

4.3 Aggregations and Element-wise Operations

Aggregations and element-wise operations are largely similar to CLA but with few extensions that leverage the design for redundancy exploitation and optimization for total execution time.

Tuple Frequencies and Defaults: Aggregations like sum($\mathbf{X}$) or colSums($\mathbf{X}$) pre-aggregate counts and then scale and accumulate the dictionary $\mathbf{D}_i$. In CLA, column group types like OLE only need to aggregate segment sizes, but DDC column groups still required a full scan of the index structures. In *Aware* the most common column groups are DDC and SDC. By materializing the tuple frequencies, we can often reuse $\mathbf{P}_i$, yielding better asymptotic behavior. For matrix-scalar and matrix/row-vector operations—as used for standardization (e.g., $\mathbf{X} - \text{colMeans}(\mathbf{X})$—we further preserve the structural redundancy by handling default values in SDC column groups (e.g., replace zero by column mean), leaving the index structures unchanged.

Handling Overlapped State: After right matrix multiplications, operations have to deal with—but can also leverage and propagate—overlapping state with partial sum semantics. Generally, sum-product operations can be executed directly on overlapping state. While left matrix multiplications directly apply, aggregations and element-wise operations require special treatment. The list of extended operations includes full or column aggregations like sum or mean, but also matrix-scalar or matrix/row-vector multiplications and additions. For matrix/row-vector addition and subtraction (e.g., $\mathbf{X} - \text{colMeans}(\mathbf{X})$, we can add a single overlapping column group (of

constant columns), whereas for $\mathbf{X}/\text{colSds}(\mathbf{X})$, we process all overlapping dictionaries via uncompressed matrix-vector operations. Operations like min/max/pow or matrix-matrix operations are not supported in overlapping state because these types do not distribute over sums.

Decompression: Unsupported *Aware* operations or operations that cannot process overlapping state are then handled by decompression. We use cache-conscious blocking for converting the column-compressed matrix into a row-major uncompressed matrix. In contrast to update-fragment compaction during read in TileDB [59] (latest writer wins), we accrue the overlapping column group contributions into the output with $O(nm|\mathcal{G}|)$ time complexity (worst-case of $|\mathcal{G}|$ overlapping groups). Although decompression is expensive, it ensures robustness and allows controlling potential redundancy, but if more efficient than full decompressions, we support partial decompressions to enable the operations to process sub-parts of the matrix at a time.

5 WORKLOAD-AWARE COMPRESSION

Aware aims to achieve broad applicability by redundancy exploitation and optimization for execution time. Instead of compressing input matrices only according to data characteristics, we extract workload characteristics from the given LA program, and compress candidate inputs and intermediates in a data- *and* workload-aware manner (Section 5.1), and then leverage compressed data characteristics for a refined compilation of execution plans (Section 5.2).

5.1 Workload Trees and Compression

Given a linear algebra program, workload-aware compression selects intermediates as compression candidates, and for each candidate extracts a workload tree (a compact workload summary seen in Figure 5), evaluates its costs, and if valid for compression, injects a `compress` directive that utilizes the workload for fine-tuned (i.e., workload-aware) compression.

Workload Trees: Many workloads in practice are complex LA programs with conditional control flow, non-trivial function call graphs, and thousands of operations. However, compressing an input or intermediate often affects only a small subset of data-dependent operations. We introduce the notion of a *workload tree* as a compact representation of these operations to simplify optimization. A workload tree for a single candidate intermediate represents the program hierarchy of conditional control flow (branches and loops) as well as function calls as inner nodes, and relevant compressed operations as leaf nodes. Here, parent-child relationships represent containment. For the sake of compactness, the tree comprises only inner nodes that contain at least one compressed operation. Counting frequencies and costing is then an aggregation across hierarchy levels. For loops, we multiply the costs by the number of iterations. If the number of iterations is unknown (e.g., convergence-based loops), we assume a constant 10 to reflect that operations inside the loop, are likely executed multiple times. Some instructions are further multiplied by the dimensionality of the inputs, and if unknown during optimization, a multiplier of 16.

Workload Tree Extraction: Our initial candidate selection and optimization approach relies on heuristics. We make a linear scan over the program, and extract compression candidates by operation type (e.g., persistent reads, comparisons, `ctable`, and rounding) as well as shape constraints (dimensions, and row/column ratios). Together, these heuristics find good candidates while keeping the number of candidates low. For each candidate, we then make a scan over the program and extract its workload tree by computing the transitive closure of derived compressed intermediates (based on operation types that are known to produce compressed outputs). Again in a heuristic manner, we then evaluate individual candidates independently without considering joint effects of groups of compressed intermediates. This extraction also descents into functions, but via stack-based identification includes only the first level of recursive function calls. In this context, we prune the unnecessary extraction of workload trees for overlapping intermediates. We perform this

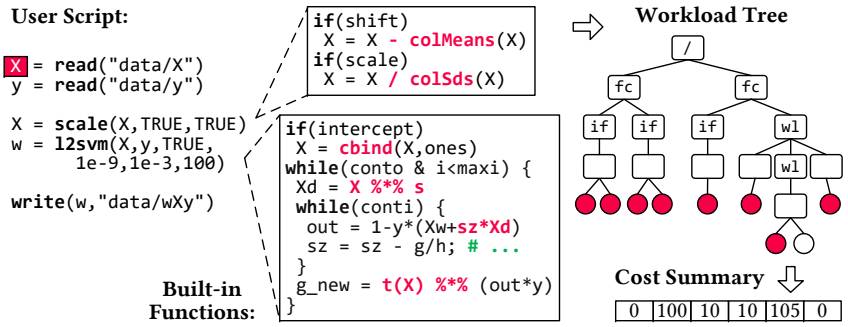


Fig. 5. Example showing a user script that reads a feature matrix $\mathbf{X}$ and label vector $\mathbf{y}$, normalizes $\mathbf{X}$ to mean 0 and standard deviation 1, and then trains an l2-regularized support vector machine. These functions are themselves linear-algebra scripts. Assuming $\mathbf{X}$ as a compression candidate, we extract the workload tree at the right, which contains 2 function calls, 3 if branches, 2 nested while loops and 8 compressed operations and 1 decompression. Aggregating the workload tree yields a cost summary for categories of operations.

extraction at the end of inter-procedural analysis (IPA). At this point, literals and size information have been propagated across the program and into amenable functions, and many simplifications have been performed. In Figure 5, we would have propagated the shift, scale, and intercept flags, removed unnecessary branches, and inlined the scale function into the main program.

Cost Evaluation: The cost summaries computed from the workload tree serve two purposes: for comparing uncompressed operations, and for guiding compression planning. We compute both frequencies and costs, where the latter utilizes the cost functions from Table 3. We organize the cost summaries by categories of operations with different behavior in compressed space: (0) Decompression, (1) Overlapping Decompression, (2) LMM, (3) RMM, (4) TSMM, (5) Dictionary-Ops, and (6) Indexing-Ops. Decompression is the frequency of regular decompressions, while overlapping decompression converts the overlapping output into uncompressed form if operations are not applicable on partial aggregates; both counts are multiplied by number of columns in each occurrence. LMM is multiplied by the number of rows on the left, RMM multiplied by the number of columns on the right, and TSMM includes counts of compressed multiplications and transpose-self multiplications and is multiplied by the number of columns. Dictionary operations can be performed directly on the compressed dictionaries (e.g., sum or element-wise scalar operations). Finally, Indexing refers to slicing of batches or blocking during broadcasting. If the cost evaluation suggests that compressing an intermediate may be beneficial, we make the cost summary globally available, and inject the related `compress` directive.

Compression Planning: The `compress` directives injected into the execution plan, perform compression as described in Section 3.2, but for workload-awareness get a cost summary. The workload mix influence the selection of column group types, co-coding decisions, and tuning for compression ratios. During classification and co-coding, we estimate the column costs from the sample, and then use these costs to decide on column groupings instead of grouping purely for compression ratios. However, including I/O costs also enables adapting the compression plans for large out-of-core datasets where good compression ratios are important to fit data in memory and/or reduce I/O. Local compression directly leverages the cost summaries, while for distributed compression, we serialize the cost summaries and compress blocks independently.

Adaptive Compression: In our federated learning backend [8, 9], standing worker processes execute continuously incoming operation requests from multiple tenants. *AWARE* also supports these dynamically changing workloads by summarizing cost vectors of previously executed operations and triggering asynchronous compression to adapt the compressed representation when needed.

5.2 Workload-aware Execution Planning

In the context of hybrid runtime plans—composed of local in-memory, and distributed Spark operations—after compression, opportunities for compiling more efficient plans arise. Specifically, we can compile operations on compressed intermediates to local operations, where uncompressed operations would exceed the memory budget and fallback to distributed runtime plans.

Program Restructuring: The challenge is that compression happens at runtime, and thus, the estimated and actual compressed size is unknown during initial compilation. Accordingly, we create—similar to data-dependent operators with unknown output shapes [15]—artificial recompilation opportunities by splitting basic blocks after injected `compress` directives. If a block contains multiple, independent `compress` operators, we create a single cut for all.

Compression-aware Recompilation: If an operator is marked for distributed operations due to unknown input/output dimensions or sparsity during initial compilation, the entire DAG (basic block) is marked for recompilation during runtime. Workload-aware compression leverages this infrastructure for obtaining the actual size of compressed in-memory matrices, and propagating the compressed size bottom-up through the DAG. With this updated size information, we can compile and execute refined partial execution plans. Affected decisions include selected execution types (local vs Spark), and physical operators including broadcasting.

6 EXPERIMENTS

Our experiments study *Aware* in Apache SystemDS[3] in comparison with uncompressed operations (ULA), compressed linear algebra (CLA) [25, 26] in Apache SystemML, and different data types in TensorFlow. We evaluate a variety of micro benchmarks, end-to-end ML algorithms, and hyper-parameter tuning; with local, distributed, and hybrid runtime plans.

6.1 Experimental Setting

Hardware Setup: Our local and distributed experiments use a scale-out cluster of 1 + 6 (1 + 11) nodes, each having a single AMD EPYC 7443P CPU at 2.85 GHz (24 physical/48 virtual cores), 256 GB DDR4 RAM at 3.2 GHz, 1 × 480 GB SATA SSD, 8 × 2 TB SATA HDDs (data) and Mellanox ConnectX-6 HDR/200 Gb Infiniband. We use Ubuntu 20.04.1, OpenJDK Java 11.0.13 with JVM arguments `-Xmx110g -Xms110g -Xmn11g`, Apache Hadoop 3.3.4, and Apache Spark 3.2.0. The CLA baseline uses SystemML 1.2 with Spark 2.4 and equivalent configurations. Some experiments marked with * were run on another cluster (for comparison) of 1 + 6 nodes with AMD EPYC 7302 CPU at 3.0 − 3.3 GHz (16/32 cores). 128 GB DDR4 RAM at 2.933 GHz, 2 × 480 GB SATA SSDs (system/home), 12 × 2 TB HDDs (data), and 2 × 10Gb Ethernet.

Datasets: Since compression is strongly data-dependent, we exclusively use the real datasets shown in Table 4. This selection includes dense, sparse, and ultra-sparse datasets with common data characteristics. All reported sizes and compression ratios refer to the size in memory using a sparsity threshold of 0.4 for uncompressed matrices. US Census [24] is further used in an encoded form with binning/one-hot encoding for numerical, and recoding/one-hot encoding for categorical features, resulting in an increase from 68 to 378 columns, and the increased sparsity from 0.43 to 0.18, but with negligible change of the size in memory. For large-scale experiments, we use a replicated versions of US Census Enc (up to 128x) which is roughly 290 GB and after densifying operations more than 950 GB. The Spark default configuration uses a storage fraction of 0.5, which gives an aggregate cluster memory of $6 \cdot 105\,\text{GB} \cdot 0.5 = 315\,\text{GB}$. That way, we scale to data sizes that require I/O per iteration in uncompressed representation.

[3]All code and experiments are available open source in Apache SystemDS (https://github.com/apache/systemds) and our reproducibility repository (https://github.com/damslab/reproducibility).

Table 4. Datasets (n Rows, m Columns, sp Sparsity).

Dataset	n (nrow(**X**))	m (ncol(**X**))	sp	Size
Airline78 [6]	14,462,943	29	0.54	3.4 GB
Amazon [32, 55]	8,026,324	2,330,066	1.2e-6	1.22 GB
Covtype [24]	581,012	54	0.22	127 MB
Mnist1m [18]	1,000,000	784	0.25	2.46 GB
US Census [24]	2,458,285	68 (378)	0.43 (0.18)	1.34 GB
US Census 128x	314,660,480	68 (378)	0.43 (0.18)	289.5 GB

Table 5. Local Compression Times [Seconds] and Ratios.

Dataset	CLA		*AWARE*-Mem		*AWARE*	
	time	ratio	time	ratio	time	ratio
Airline78	9.34 sec	10.22	1.74 sec	8.61	2.08 sec	7.94
Amazon	*37.6 hours Crash*		8.54 sec	1.73	3.77 sec	*Abort*
Covtype	1.10 sec	13.79	0.84 sec	14.24	1.23 sec	13.99
Mnist1m	7.25 sec	7.14	4.57 sec	6.09	17.50 sec	4.41
US Census	5.15 sec	35.38	1.16 sec	29.60	1.15 sec	27.35
US Census Enc	27.48 sec	41.03	5.78 sec	38.46	6.54 sec	29.46

6.2 Compression Performance

We first investigate the compression process itself in terms of compression times, compression ratios, and the influence of workload characteristics.

Compression Ratios: Starting with local single-node matrix compression, we compare *AWARE* optimized for memory (*AWARE*-Mem) and for workload (*AWARE*) with the existing CLA framework [25, 26]. The used workload is a fixed cost summary of left matrix multiplications that leads to extensive co-coding of columns. Table 5 shows the compression times and ratios for all datasets, where the ratios are calculated from the sizes of in-memory representations. We attribute the minor differences to published CLA compression ratios [26] (Airline78 7.44, Covtype 18.19, US Census 35.69) to different data preparation and sparse memory estimates. Compared to CLA, *AWARE*yields worse compression ratios except for CovType, where *AWARE*'s co-coding finds other column groups and uses new encoding types. The lower compression ratios are due to the tuning for operations performance rather than size, and in practice, moderate absolute differences of large compression ratios have only little impact on size. For example, compressing a 1 GB input with ratio 7 versus 6 yields only a difference of 24 MB. The Amazon dataset is an interesting case, where CLA runs out of memory due to group memoization during co-coding ($> 2.7 \cdot 10^{12}$ pairs of columns). In contrast, *AWARE* aborts the compression early because the estimated total costs exceed the costs of ULA. Optimizing for memory in *AWARE* yields a low compression ratio for Amazon because of object overheads per column group, which do not exist in ULA's CSR representation.

Compression Times: Table 5 further shows the compression times for all datasets. *AWARE* generally reduces compression times when optimizing for size (up to 4.7x) and cost (up to 4.2x), which makes compression easier to amortize. The speed difference in *AWARE*-Mem US Census Enc is due to a reduction of the grouping phase from 19 to 2.4 seconds. When using workload-aware compression with a fixed cost summary that causes more grouping and compression, only MNIST and US Census have significantly slower compression compared to optimizing for memory. MNIST is slow because combining column groups have a large number of distinct values (each column contains up to 256 distinct values, and three columns together has up to 256^3 distinct tuples).

Table 6. Spark RDD Compression (Data: US Census Enc).

	Aware-Mem		*Aware*		
Blocksize	Ratio	Total Size	Ratio	Total Size	ULA
1K	8.94	225.58 MB	7.83	257.36 MB	2.02 GB
2K	15.1	143.49 MB	10.9	198.55 MB	2.17 GB
16K	26.6	81.68 MB	23.3	93.36 MB	2.17 GB
64K	29.9	72.74 MB	23.4	92.96 MB	2.17 GB
256K	30.5	71.22 MB	24.5	88.70 MB	2.17 GB

Compression Ratio Spark: For distributed operations, matrices are represented as Spark resilient distributed datasets (RDD) [83]—i.e., distributed collections of key-value pairs—with values being fixed-size matrix blocks of size $b \times b$ (except boundary blocks). These blocks are compressed independently with separate compression plans and dictionaries. The default block size is $b = 1K$ (8 MB dense blocks), but sparsity and compression warrant larger blocks. Table 6 varies this block size b for US Census, and reports the size of *Aware*-Mem, *Aware*, and uncompressed (ULA) RDDs (from Spark's cached RDD-infos). With small blocks, there is larger variability of compression, and increasing block sizes give better ratios while also stabilizing the resulting compression plans. Overall, *Aware* yields good compression ratios even with small b and approaches local compression ratios with larger b. For the remaining experiments, we use a block size of $b = 16K$.

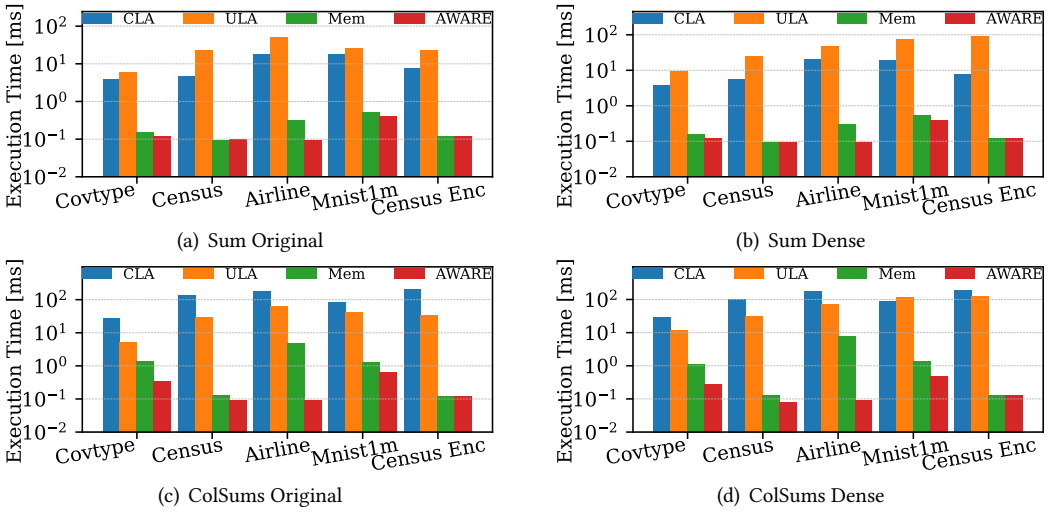
6.3 Operations Performance

In a second series of micro-benchmarks, we compare the runtime of *Aware* with ULA and CLA operations. While CLA was designed for operations performance close to uncompressed and benefits from keeping large datasets in memory, *Aware* aims to improve performance more generally, even for in-memory settings and keep performance stable even if the input data is densified. We compare using both original and densified by simply adding 1 to all cells.

Aggregations: Figure 6(a) shows the results for computing the aggregate sum($\mathbf{X}$). CLA processes each column group in parallel, aggregates individual sums, and combines them into the result. ULA uses multi-threading with sequential scans of row partitions. By memorizing the frequencies of tuples, *Aware* executes purely on the column group dictionaries without scanning their indexes because of memoization. Without memoization, the execution time increase according to the complexity of scanning the index structures. For instance, we observed no penalty in case of Census enc, while 60x performance drop in Census for *Aware*. When densifying, in Figure 6(b), CLA and *Aware* retain their performance while ULA's performance worsen. For column aggregations (Figure 6(c) and 6(d)), CLA is slower than ULA because CLA's DDC colSums is not specialized for DDC1 and DDC2 and thus, performs a lookup of dictionary values for each encoded cell.

Element-wise Operations: Figure 7(a) shows the performance of adding a scalar value to each matrix cell. We observe extreme speedups of up to 10,766x because *Aware* avoids modifying dictionaries where possible. Figures 7(b) shows similar improvements for matrix-vector (row vector) element-wise operations. Specifically, we analyze $\mathbf{X}/\mathbf{v}$ on sparse (not shown) and dense representations, where we chose division because it forces modifications of the dictionaries. We still see speedups of about three orders of magnitude (2,047x).

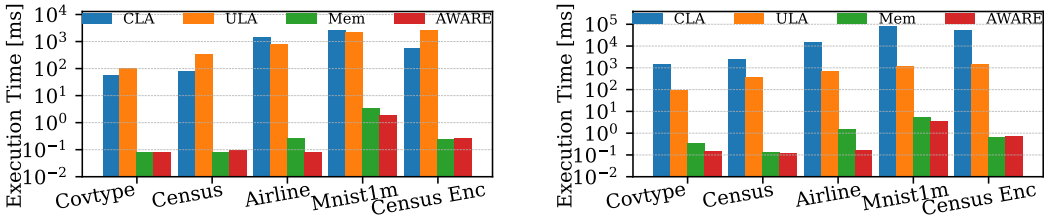
Left Matrix Multiplication (LMM): Left, right, and transpose-self matrix multiplications are key operations in many ML algorithms. In the following, we first evaluate these operations independently. Figure 8(a) shows the results of left matrix multiplications for all datasets, where the uncompressed left-hand-side has 16 rows. CLA emulates this matrix-matrix multiplication via 16 vector-matrix multiplications. We observe *Aware* performance comparable to multi-threaded



(a) Sum Original (b) Sum Dense

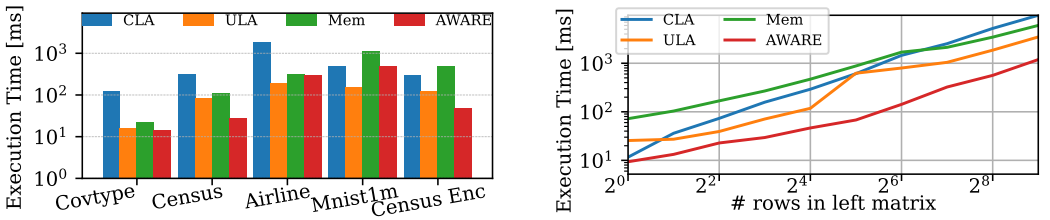
(c) ColSums Original (d) ColSums Dense

Fig. 6. Operations Performance Aggregate.



(a) Plus Scalar, Original (b) Div Row-vector, Dense

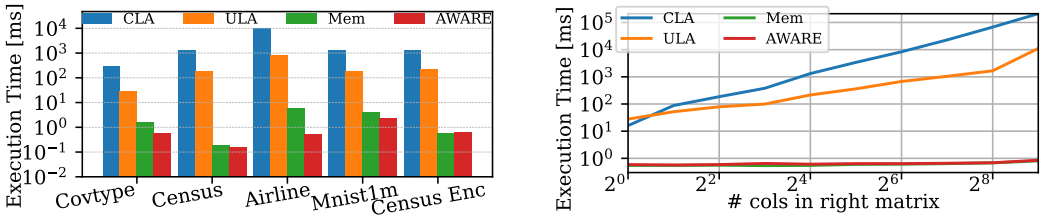
Fig. 7. Operations Performance Scalar.



(a) 16 row LMM (b) LMM Census Enc Scaling

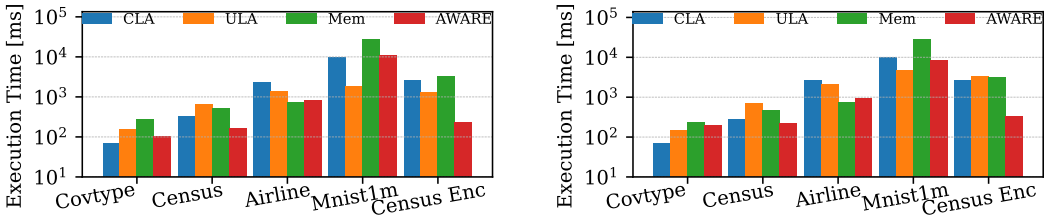
Fig. 8. Operations Performance Left Matrix Multiplication.

ULA (sparse and dense) with improvements for Covtype, US Census, and US Census Enc, but a moderate slowdown for Airlines and a significant slowdown for MNIST. LMM also shows a major performance difference when optimizing for memory versus optimizing for operations, which is especially noticeable in US Census Enc. In contrast, for datasets with smaller potential for co-coding like Airline, there is no difference. Figure 8(b) shows results on US Census Enc with varying number of rows in the left-hand-side. CLA performs similar to *Aware* at a single row, but when rows increase CLA's performance decrease to the same as ULA due to the lack of native matrix-matrix support. CLA is worse at utilizing more threads, while *Aware* and ULA scale better. For ULA and *Aware*-Mem, there is a change in parallelization strategies after 16 rows. In contrast, *Aware* yields between half and one order of magnitude speedups for all #rows configurations.



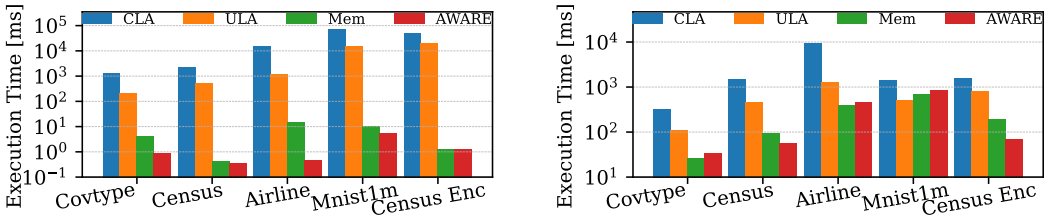
(a) 16 col RMM (b) RMM Census Enc Scaling

Fig. 9. Operations Performance Right Matrix Multiplication.



(a) TSMM Sparse (b) TSMM Dense

Fig. 10. Operations Performance Transpose Self Matrix Multiplication.



(a) Scale and Shift Sparse (b) Euclidean MinDist 16 Points Dense

Fig. 11. Operations Performance Sequence.

Right Matrix Multiplication (RMM): In contrast to LMM, the right matrix multiplication creates outputs of overlapping column groups with a shallow copy of the index structures. Figure 9(a) shows the results for all datasets, where we observe *AWARE* speedups between 53x to 1,528x because of the deferred aggregation across column groups. Figure 9(b) then shows the scaling with increasing number of columns in the uncompressed right-hand-side. CLA shows equal performance to uncompressed in the single column case but scales worse then ULA, again due to the lack of native matrix-matrix multiplication. *AWARE*'s RMM exhibits better asymptotic behavior due to its dictionary-centric operations, yielding speedups >13,000x for 512 columns.

Transpose-Self Matrix Multiplication (TSMM): Figures 10(a) and 10(b) show the results of TSMM operations as used for computing PCA, direct-solve linear regression, as well as covariance and correlation matrices. We observe speedups on all datasets except MNIST, where *AWARE* yields a substantial slowdown, especially for sparse inputs. The TSMM performance is largely dependent on the number of column groups, their number of distinct items, and thus, co-coding decisions. MNIST has a high number of columns, with high cardinality, and low correlation between columns.

Operation Sequences: As final micro benchmark use cases, we evaluate two sequences of operations. First, *scale and shift* in Figure 11(a) performs a shifting $\mathbf{Y} = \mathbf{X} - (\text{colSums}(\mathbf{X})/\text{nrow}(\mathbf{X}))$ and scaling $\mathbf{Z} = \mathbf{Y}/\sqrt{\text{colSums}(\mathbf{Y}^2)/(\text{nrow}(\mathbf{Y}) - 1)}$. This sequence is a common normalization step (standard-scaler) of the input data but has the negative side effect of densifying the input data.

Table 7. RMM Overlap Sequence (Data: US Census Enc).

	I/O	Comp	RMM	Total
SystemML - ULA	0.84 sec	—	188.40 sec	190.03 sec
SystemML - CLA	0.88 sec	24.34 sec	374.13 sec	401.27 sec
SystemDS - ULA	0.81 sec	—	189.27 sec	190.42 sec
Aware-No OL	0.76 sec	3.97 sec	189.59 sec	195.51 sec
Aware-Mem	0.80 sec	8.00 sec	0.38 sec	9.72 sec
Aware	0.78 sec	3.93 sec	0.42 sec	5.69 sec

Aware improves performance up to a best case of 15,399x. Second, we compute the minimum Euclidean distances via $\mathbf{D} = -2 * (\mathbf{X} \times \mathrm{t}(\mathbf{C})) + \mathrm{t}(\mathrm{rowSums}(\mathbf{C}^2))$, followed by $\mathbf{d} = \mathrm{rowMins}(\mathbf{D})$ (which forces a decompression from overlapping state). Here, $\mathbf{D}$ are the Euclidean distances of each row in $\mathbf{X}$ to the centroids $\mathbf{C}$. This expressions is used, for instance, in K-Means clustering. *Aware* shows performance up to 11.3x faster compared to ULA in all cases except MNIST.

Overlap: Leveraging the overlapping output from RMM without compaction shows significant improvements in Figures 9(a) and 11. However, overlapping representations are most beneficial in chains of RMMs. Table 7 shows the end-to-end runtime for a sequence of 10 RMM of size $k = 512$, representative for processing 10 fully-connected layers of size 512 with no activation. CLA is slower than ULA in this scenario because it is falling back to vector matrix compressed operations for the first multiplication. *Aware* with no overlapping is slower because the first right multiplication decompress, but it does show close to ULA performance. *Aware* with overlapping column groups push the compressed index structures through the entire chain of RMMs, improving performance irregardless of optimizing for memory or workload, with a slight advantage to workload.

Computation Cost: Table 8 shows the *Aware* workload analysis of different micro benchmarks on Census_Enc. This experiment shows the estimated Theoretical Operations (TOPS), calculated from the cost vectors and compression schemes. We compare the estimated TOPS for uncompressed operations (on the left) with *Aware*'s estimated TOPS extracted from the sample and co-coding decisions, as well as the estimated TOPS after compression (on the right). We observe that the estimated TOPS from the sample is close to the actual TOPS, indicating good estimation accuracy and thus, meaningful costs. We also show the compression time (Comp) and the runtime (Time) for executing 100 repetitions of the given operation (Op 100×). There are some micro benchmarks that show disproportionate scaling of runtimes compared to TOPS. With small execution times, moderate discrepancies are expected because of various unaccounted overheads in both ULA and *Aware*. For TSMM and LMM, the differences are due to output allocation, memory bandwidth limitations, and index structure lookups. Although the runtime discrepancies are sub-par, we found

Table 8. *Aware* Workload TOPS (Data: US Census Enc).

	ULA		*Aware*			
Op (100×)	TOPS	Time	Est. TOPS	TOPS	Comp	Time
SUM	3.38e+10	2.25 sec	1.29e+05	1.14e+05	4.60 sec	0.08 sec
SUM Dense	1.90e+11	8.96 sec	1.31e+05	1.14e+05	4.65 sec	0.07 sec
RMM-256	2.81e+13	156.97 sec	2.13e+07	1.94e+07	4.74 sec	0.25 sec
LMM-256	4.28e+12	185.69 sec	6.87e+11	7.14e+11	7.22 sec	53.76 sec
TSMM	6.32e+12	111.12 sec	9.83e+11	9.98e+11	7.19 sec	16.91 sec
ScaleShift	7.47e+11	3,216.21 sec	4.08e+05	3.42e+05	4.89 sec	0.36 sec
Euclidean-256	4.80e+13	308.85 sec	8.61e+11	9.04e+11	7.87 sec	78.55 sec

Table 9. Workload-awareness on Local End-to-End Algorithms (Data: US Census Enc)

	ULA	***Aware*-Mem**		***Aware***	
	Time	Comp	Time	Comp	Time
K-Means	51.6 sec	4.2 sec	46.2 sec	6.2 sec	27.1 sec
PCA	12.7 sec	4.0 sec	10.4 sec	6.0 sec	9.0 sec
MLogReg	32.0 sec	4.5 sec	32.5 sec	7.2 sec	26.0 sec
lmCG	19.8 sec	5.0 sec	20.7 sec	6.4 sec	18.6 sec
lmDS	15.6 sec	5.7 sec	15.5 sec	6.1 sec	14.3 sec
L2SVM	38.9 sec	6.5 sec	45.2 sec	6.2 sec	36.5 sec

Table 10. L2SVM (without scale&shift, 60 iterations, Data: US Census Enc)

	Local (1x)				Distributed (256x)
	I/O	**Comp**	**L2SVM**	**Total**	**Total**
SystemML - ULA	1.6 sec	—	36.7 sec	38.4 sec	5,689.6 sec
SystemML - CLA	1.5 sec	32.8 sec	31.7 sec	66.0 sec	4,722.7 sec
SystemDS - ULA	1.6 sec	—	19.3 sec	20.9 sec	2,849.1 sec
Aware-Mem	1.4 sec	6.0 sec	21.3 sec	28.7 sec	2,300.4 sec
Aware	1.6 sec	7.9 sec	15.9 sec	25.3 sec	2,294.9 sec

that our TOPS estimation provides a good balance of simplicity and reflecting key differences relevant for compression. More sophisticated cost estimators are, however, interesting future work.

6.4 End-to-End Algorithm Performance

We use the following six algorithms to evaluate *Aware* with workload-ware compression on end-to-end ML training: K-Means for clustering; principal component analysis (PCA) for dimensionality reduction; multinomial (multi-class) logistic regression (MLogReg); LM via conjugate gradient (lmCG), and via a direct solve method (lmDS) for linear regression; as well as l2-regularized support vector machines (L2SVM) for classification. In theory, *Aware* gives equal results to ULA but because of rounding errors in sequences of FP64 operations and different parallelization strategies—present both, in ULA and *Aware*—algorithms naturally execute with slight variations. Therefore, algorithm parameters are set to ensure an equal number of iterations and operations. We use the US Census Enc dataset and scale up by replication. The replication maintains the statistics of the data, and is not an issue for distributed execution, where blocks are compressed independently. The local influence is limited to constant dictionary sizes, and replication is not actively exploited by *Aware*.

Local Execution: Table 9 shows the results algorithms fit in-memory. *Aware* yields moderate but consistent improvements, or at worst (e.g. L2SVM, lmDS) comparable performance. Observing improvements on all algorithms most notably 19% for MLogReg, 47% for K-Means (iterative algorithms), and 29% for PCA (non-iterative algorithm) is remarkably because this includes online compression. Underlying reasons are fast compression that is easier to amortize and redundancy exploiting operations. The algorithms L2SVM, lmCG, and lmDS all perform very close to ULA.

CLA Comparison: CLA is not included in Table 9 because it does not support scale&shift and therefore would not execute efficiently. For a fair comparison, we use the L2SVM algorithm from CLA [26] (with minor modifications, e.g. 60 iterations not 100) and compare different configurations of CLA (in SystemML) and *Aware* in Table 10. Both systems read and parse both train and test datasets (in binary), increasing I/O compared to the other experiments. We observe that CLA compression is slower than *Aware* optimizing for size or compute. CLA does not outperform

Table 11. End-to-End Algorithms Hybrid Execution [Seconds] (Data: US Census Enc, *D* .. Incl. Distributed Ops).

	K-Means		PCA		MLogReg	
	ULA	***Aware***	**ULA**	***Aware***	**ULA**	***Aware***
1x	51.6	(6) 27.1	12.7	(6) 9.4	32.0	(7) 26.0
8x	471.0	(26) 117.8	330.3	(26) 42.6	393.3	(29) 88.2
16x	D484.3	(48) 183.9	D76.3	(47) 67.5	D570.3	(58) 144.2
32x	D1,491.6	D1,496.3	D70.3	D61.2	D671.5	D629.9
128x	D17,819.0	D6,298.0	D137.0	D140.3	D3,502.9	D1,710.6
*128x	D33,039.0	D11,616.0	D269.0	D259.0	D50,998.0	D8,599.6
	lmCG		**lmDS**		**L2SVM**	
	ULA	***Aware***	**ULA**	***Aware***	**ULA**	***Aware***
1x	19.8	(6) 18.6	15.6	(6) 14.3	38.9	(6) 36.5
8x	366.2	(26) 60.6	334.4	(29) 51.5	405.2	(26) 115.4
16x	D104.4	(44) 91.7	D80.2	(50) 75.8	D252.6	(56) 195.5
32x	D264.6	D105.3	D91.5	D70.8	D433.2	D479.4
128x	D1,611.4	D242.6	D175.9	D162.4	D5,286.9	D1,904.5
*128x	D33,090.0	D469.0	D365.9	D465.0	D74,016.0	D1,060.0

ULA in SystemML in local settings because the compression is not amortized. In contrast, our ULA baseline is 1.9x faster, *Aware*-Mem shows similar performance to CLA, and *Aware* improves the relative training time (without compression and IO) by 2x over CLA, and 2.3x over ULA, but SystemDS ULA is the fastest end-to-end. Since CLA mostly focuses on large distributed datasets, we further compare CLA and *Aware* on a larger sparse dataset (256x, which only partly fits in memory of 11 nodes). Table 10 (right) shows that SystemML CLA yields a moderate speedup, but *Aware* achieves another 2x over SystemML CLA. At this scale, *Aware* optimizes for memory size and thus, the results *Aware* and *Aware*-Mem are similar.

Hybrid Execution: In between the local and distributed extremes, there are hybrid runtime plans, where the sparse input fits into memory of the driver but after scale&shift transformation, the transformed data does not fit in the driver and thus, generates distributed operations. Table 11 show the results for replicated versions of US Census Enc (8x-32x). Runs using distributed operations are marked with *D* and local compression times are included in parenthesis. These in-between scenarios are generally challenging in terms of evictions, efficient exchange between local and distributed runtimes, as well as decisions on when to prefer distributed operations. Most notable is this characteristic in ULA, which is sometimes faster for larger scales. This is because the execution fits in memory for various instructions and therefore more or less instructions are executed distributed. For instance in PCA, the number of distributed instructions grows from 16x to 32x to finally 128x. Across all algorithms—except for a few instances—*Aware* show consistent improvements, especially if we focus on computation time (without the compression time).

Large-scale Execution: Finally, the last two rows (128x) show the primary compression scenario, where both the sparse input and dense intermediate after transformation do not fit into local memory and the dense intermediate exceeds aggregate cluster memory. We still compile hybrid runtime plans but all operations on **X** (and some derived intermediates) are distributed. Since the data exceeds aggregate memory, iterative algorithms read in every iteration more than two thirds of **X** from evicted partitions. The 128x results refer to our primary cluster setup but with a different

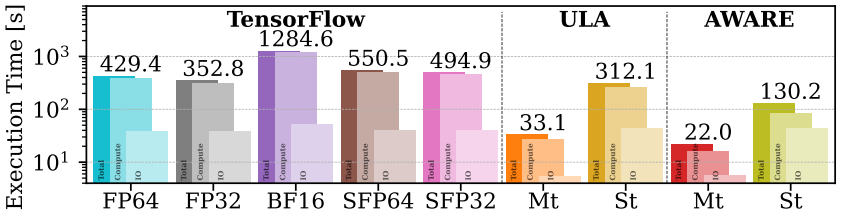


Fig. 12. TensorFlow Comparison (lmCG, US Census Enc).

memory configuration (more executors and nodes, smaller `spark.memory.fraction`) in order to ensure stable results. For comparison, we also include previous results from our secondary cluster (128x*) using the same configuration as hybrid execution, which caused lost executors in some cases. Due to redundancy exploitation and good compression ratios—even on tiles (see Table 6)—we observe large improvements of 2.8x for K-Means, 2x for MLogReg, 6.6x for lmCG, and 2.8x for L2SVM. In contrast, PCA and lmDS are non-iterative algorithms. On the secondary cluster, we observed up to 70x performance gains. The differences in relative improvements are due to faster networking and OS file system caching of evicted partitions due to more physical memory per node (256 GB versus 128 GB), which favors uncompressed (ULA) operations.

6.5 Additional Baseline - TensorFlow

While ULA is the most important baseline—within the same compiler and runtime— we also compare with TensorFlow (TF) version 2.12; We evaluated both TF and TF-AutoGraph [52], but report numbers for TF-AutoGraph, which gave 1-5 sec faster execution times on average. The workload is a simplified version of lmCG on US Census Enc, expressed via TF linear algebra operations. By default, we use 300 lmCG iterations (instead of 100 in Table 9).

Results: Figure 12 shows the results in log-scale, where each stack is I/O time, compute time, and total time (from front to back, as regular stacking is infeasible in log scale). On the left, we have TF with different value types. Changing from FP64 (double) to FP32 improves execution time by 21.7%, reducing to FP16 produces infinite sums, rendering the algorithm invalid. BF16 solves this issue by using a different numbers of exponent and mantissa bits, but it is not well supported on the CPU, resulting in a 3x slowdown compared to FP64. Using TF's sparse representation worsen performance slightly at FP64 precision similarly at FP32. TF executes the core expression per iteration (of two matrix-vector multiplications) $\mathbf{X}^\top(\mathbf{X}\,\mathbf{v})$ single-threaded because it only uses multi-threaded matrix multiplications with two or more columns in right-hand-side matrix. In contrast, our multi-threaded I/O and matrix-vector multiplications yield speedups of about 13x for ULA and 19.5x for *Aware*. Forcing both single-threaded I/O and operations (St), ULA becomes 38% faster than TF. ULA (with data size of 1.3 GB) does not fully saturate the memory-bandwidth for this sparse dataset, while *Aware* fits the compressed matrix (49.7 MB) into the 128 MB L3 cache, yielding a 3.3x speedup over TF-FP64. To summarize, both ULA and *Aware* show competitive performance with single-threaded, and are faster with multi-threaded operations, indicating that *Aware*'s improvements could carry over to other ML systems.

6.6 Hyper-Parameter Tuning

Executing a single short ML training algorithm makes it hard to amortize the online compression. In practice, however, most time is spent in ML pipelines that involve outer loops for enumerating data augmentation pipelines, feature and model selection, hyper-parameter tuning, and model debugging. *Aware* adapts to such more complex workloads by spending more time on compression (which is easily amortized) and optimizing for operation performance in the inner loops. Table 12

Table 12. GridSearch MLogReg (Data: US Census Enc).

ULA	*Aware*-Mem	*Aware*
274.3 sec	238.1 sec	**92.6 sec**

shows results for a basic GridSearch hyper-parameter tuning of the MLogReg algorithm. Even for a small number of $3 \cdot 3 \cdot 3 \cdot 5 = 90$ hyper-parameter configurations, *Aware* improves the local runtime (including compression) by 3x, which is a promising result for wide practical applicability.

7 RELATED WORK

Workload-aware, lossless matrix compression is related to lossless and lossy matrix compression, query processing on compressed data, and workload-aware physical design of compressed data.

Lossless Matrix Compression: Naturally, the closest area of related work is lossless matrix compression, whose limitations we already discussed in Section 2. General-purpose data-parallel frameworks like Spark [83] or Flink [5], scientific data formats like NetCDF and HDF5 [31], and storage managers like SciDB [71] and TileDB [59] also support compression, but decompress block- or partition-wise for operations. Early work includes traditional sparse matrix representations (e.g., CSR, CSC, COO) [64] and compression techniques by Kourtis et al. that already leveraged dictionary coding [37, 40], as well as delta and run-length encoding [37]. Subsequent work on compressed linear algebra (CLA) focused on online compression and entire ML algorithms, where CLA [25, 26] uses column compression for batch algorithms, and TOC [45] uses tuple compression for mini-batch algorithms. Other works exploit different properties such as: integer time series values [12], floating point time series [46], and bounded ranges of floats [48]. Recent work on grammar-compressed matrices report operation performance proportional to the compressed size [27], while others presented impossibility results (worst-case) for efficient matrix-vector multiplications on grammar-compressed matrices such as Lempel-Ziv [4]. Factorized learning [42, 67] further pushes operations of ML training algorithms through joins and can be seen as a specialized form of lossless compression exploiting available schema information [57] to avoid materializing denormalized tables. These factorization ideas can also be implemented on top of ML systems [20] by representing joins via structured selection matrices. Compared to these mostly data-centric compression frameworks—where LMFAO [68] also compiles efficient sum-product plans for factorized learning—our *Aware* framework leverages both data and workload characteristics of linear algebra programs and adjusts the compression process, compressed representations, and execution plans accordingly.

Lossy Compression and Sampling: In the context of mini-batch DNN training and scoring, we see broad adoption of lossy compression. First, quantization discretizes floating-point into fixed-point representations such as `UINT8` for scoring [30]. Common techniques are static min/max binning (equi-width) [30] and learned quantization schemes (equi-height via quantiles) [84, 85]. Such quantization schemes are also used for efficient data transfer in ZipML [84] and SketchML [35]. With residual accumulation at the workers, some systems reduce communicated values to a single bit [69]. Second, the challenges of training with low 8-bit FP precision are addressed with chunk-based accumulation and stochastic rounding [77]. Third, there are techniques like mantissa truncation [3, 10] and new data types with different trade-offs of range and precision [56]. Examples are Google's bfloat16 (1+8+7 bits) [65], Intel's Flexpoint (shared subset of exponent bits) [39], and NVIDIA's TF32 (1+8+10) [56]. Fourth, other techniques include sparsification or value clipping (omit small values) [29, 56], dimensionality reduction like auto encoders [34], sampling in BlinkML [60], DNN activation compression in COMET [36], and progressive compression schemes [41, 78]. Unfortunately, the unknown impact on results, creates trust concerns, requires trial and error, and is problematic for declarative ML pipelines. Recent work in MLWeaving [78] introduced

data structures for efficiently extracting different granularities for simplifying exploration, while BlinkML [60] estimates the minimum sample size to satisfy an accuracy constraint. Our work on lossless compression is orthogonal as it guarantees correct results.

Workload-aware Physical Database Design: Work on lossless matrix compression like CLA [25, 26] and TOC [45] was inspired by lossless compression in column stores and related query processing on compressed data. There is a wide variety of lightweight lossless data compression schemes such as null suppression, run-length encoding, dictionary coding, frame of reference, and delta coding [1, 2]. Extensions include patched encoding schemes (separate handling of exception values) [86], order-preserving dictionary coding [11, 49], and exploitation of such schemes in query processing [7, 11, 43, 61]. Our handling of default values is also related to header compression in SAP HANA [66] and fast-mode column adds in Teradata [74]. The performance/compressed-size tradeoffs of existing schemes are, however, strongly data-dependent [22, 33]. For that reason, existing systems largely rely on conservative selection heuristics [1, 2, 43], but there is also work on cost modeling [17, 19, 22], and balancing query performance and storage size with different column group projections and encodings [75]. Once compression choices are reflected in the costs, they influence what-if physical design tuning. Compression-aware design tuning [38] showed how index compression can affect index selection choices, and learned partitioning schemes maximize partitioning pruning [82] (e.g., via small materialized aggregates [51]). Furthermore, recent work introduced memory-budget-constrained offline compression for selecting encoding schemes based on estimated costs and compression ratios [16], and related data partitioning across storage tiers [44, 76]. In contrast, our workload-aware compression planning summarizes the workload of a linear algebra program in order to tune *online* lossless matrix compression and compressed operations.

8 CONCLUSIONS

We introduced *Aware* as a workload-aware, lossless matrix compression framework with new encoding schemes and compressed operations. Compared to previous lossless matrix compression, *Aware* summarizes the workload characteristics of a linear algebra program and selects where and how to compress the inputs and intermediates for minimizing total execution time. Based on a variety of experiments, we draw two major conclusions. First, the broader spectrum of compression techniques (column groups, fast compression, overlapped representations) yields runtime improvements even when uncompressed operations fit in memory, and can handle increasingly complex ML pipelines of data preparation, model training, and debugging. Second, the workload-aware compression planning nicely adapts the compressed representation for higher compression ratios when needed, and otherwise prefers operation performance. Together, these characteristics yield a compression framework with robust performance and thus, more general applicability. Interesting future work includes the pushdown of compression into data preparation (e.g., feature transformations, and data cleaning) [81], extensions for federated learning (e.g., extended asynchronous compression) [8, 9], and combinations with lossy compression (e.g., bounded loss[36, 47]).

ACKNOWLEDGMENTS

The work on *Aware* was part of the ExDRa project, which received funding from the bilateral, German-Austrian program “ICT of the Future – Smart Data Economy" by the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK, 873838, 06/2019-08/2022). Furthermore, thanks to Patrick Damme for valuable discussions on CLA specializations, and the anonymous reviewers for their constructive reviews.

REFERENCES

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column oriented Database Systems. *PVLDB* 2, 2 (2009), 1664–1665. https://doi.org/10.14778/1687553.1687625

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682. https://doi.org/10.1145/1142473.1142548

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[4] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. 2020. Impossibility Results for Grammar-Compressed Linear Algebra. In *NeurIPS*. https://arxiv.org/abs/2010.14181

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964. https://doi.org/10.1007/s00778-014-0357-y

[6] American Statistical Association (ASA). 2009. Airline on-time performance dataset. https://stat-computing.org/dataexpo/2009/the-data.html.

[7] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. In *PVLDB*. 2393–2407. https://doi.org/10.14778/3358701.3358707

[8] Sebastian Baunsgaard, Matthias Boehm, Ankit Chaudhary, Behrouz Derakhshan, Stefan Geißelsöder, Philipp M. Grulich, Michael Hildebrand, Kevin Innerebner, Volker Markl, Claus Neubauer, Sarah Osterburg, Olga Ovcharenko, Sergey Redyuk, Tobias Rieger, Alireza Rezaei Mahdiraji, Sebastian Benjamin Wrede, and Steffen Zeuch. 2021. ExDRa: Exploratory Data Science on Federated Raw Data. In *SIGMOD*. 2450–2463. https://doi.org/10.1145/3448016.3457549

[9] Sebastian Baunsgaard, Matthias Boehm, Kevin Innerebner, Mito Kehayov, Florian Lackner, Olga Ovcharenko, Arnab Phani, Tobias Rieger, David Weissteiner, and Sebastian Benjamin Wrede. 2022. Federated Data Preparation, Learning, and Debugging in Apache SystemDS. In *CIKM*. 4813–4817. https://doi.org/10.1145/3511808.3557162

[10] Souvik Bhattacherjee, Amol Deshpande, and Alan Sussman. 2014. PStore: an efficient storage framework for managing scientific data. In *SSDBM*. 25:1–25:12. https://doi.org/10.1145/2618243.2618268

[11] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. 283–296. https://doi.org/10.14778/2536222.2536233

[12] Davis W. Blalock, Samuel Madden, and John V. Guttag. 2018. Sprintz: Time Series Compression for the Internet of Things. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 3 (2018), 93:1–93:23. https://doi.org/10.1145/3264903

[13] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[14] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768. https://doi.org/10.14778/3229863.3229865

[15] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62.

[16] Martin Boissier. 2022. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. *PVLDB* 15, 4, 780–793. https://doi.org/10.14778/3503585.3503588

[17] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*. 674–677. https://doi.org/10.5441/002/edbt.2019.84

[18] Léon Bottou and Gaëlle Loosli. 2007. The infinite MNIST dataset. https://leon.bottou.org/projects/infimnist.

[19] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *aiDM@SIGMOD*. 32–35. https://doi.org/10.1145/3464509.3464885

[20] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *PVLDB* 10, 11 (2017), 1214–1225. https://doi.org/10.14778/3137628.3137633

[21] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. 72–83. https://doi.org/10.5441/002/edbt.2017.08

[22] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46. https://doi.org/10.1145/3323991

[23] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. In *PVLDB*. 2396–2410. https://doi.org/10.14778/3407790.3407833

[24] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml

[25] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB* 9, 12 (2016), 960–971. https://doi.org/10.14778/2994509.2994515

[26] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2018. Compressed linear algebra for large-scale machine learning. *VLDB J.* 27, 5 (2018), 719–744. https://doi.org/10.1145/3318221

[27] Paolo Ferragina, Travis Gagie, Dominik Köppl, Giovanni Manzini, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. 2022. Improving Matrix-vector Multiplication via Lossless Grammar-Compressed Matrices. *CoRR* abs/2203.14540 (2022). https://doi.org/10.14778/3547305.3547321

[28] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. 2021. Pruning Neural Networks at Initialization: Why Are We Missing the Mark?. In *ICLR*. https://openreview.net/forum?id=Ig-VyQc-MLK

[29] Google. 2019. TensorFlow Model Optimization Toolkit - Pruning API. https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html.

[30] Google. 2020. Quantization Aware Training with TensorFlow Model Optimization Toolkit - Performance with Accuracy. https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html.

[31] The HDF Group. 2021. The HDF5 Library and File Format. https://www.hdfgroup.org/solutions/hdf5/.

[32] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*. 507–517. https://doi.org/10.1145/2872427.2883037

[33] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. 2007. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*. 389–400. https://doi.org/10.1145/1247480.1247525

[34] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *SIGMOD*. 1733–1746. https://doi.org/10.1145/3318464.3389734

[35] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. 2018. SketchML: Accelerating Distributed Machine Learning with Data Sketches. In *SIGMOD*. 1269–1284. https://doi.org/10.1145/3183713.3196894

[36] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. 2022. COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression. In *PVLDB*. 886–899. https://doi.org/10.14778/3503585.3503597

[37] Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. 2013. An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication. *IEEE Trans. Parallel Distributed Syst.* 24, 10 (2013), 1930–1940. https://doi.org/10.1109/TPDS.2012.290

[38] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. 2011. Compression Aware Physical Database Design. *PVLDB* 4, 10 (2011), 657–668. https://doi.org/10.14778/2021017.2021023

[39] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NeurIPS*. 1742–1752. https://proceedings.neurips.cc/paper/2017/hash/a0160709701140704575d499c997b6ca-Abstract.html

[40] Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. 2008. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF*. 87–96. https://doi.org/10.1145/1366230.1366244

[41] Michael Kuchnik, George Amvrosiadis, and Virginia Smith. 2021. Progressive Compressed Records: Taking a Byte out of DeepLearning Data. In *PVLDB*. 2627–2641. https://doi.org/10.14778/3476249.3476308

[42] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*. 1969–1984. https://doi.org/10.1145/2723372.2723713

[43] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326. https://doi.org/10.1145/2882903.2882925

[44] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. 2021. Workload-Driven Placement of Column-Store Data Structures on DRAM and NVM. In *DaMoN@SIGMOD Workshop*. 5:1–5:8. https://doi.org/10.1145/3465998.3466008

[45] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. 2019. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD*. 1517–1534. https://doi.org/10.1145/3299869.3300070

[46] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. In *PVLDB*. 3058–3070. https://doi.org/10.14778/3551793.3551852

[47] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. 2020. Plato: Approximate Analytics over Compressed Time Series with Tight Deterministic Error Guarantees. In *PVLDB*. 1105–1118. https://doi.org/10.14778/3384345.3384357

[48] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. 2021. Decomposed Bounded Floats for Fast Compression and Queries. In *PVLDB*. 2586–2598. https://doi.org/10.14778/3476249.3476305
[49] Chunwei Liu, McKade Umbenhower, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. 2019. Mostly Order Preserving Dictionaries. In *ICDE*. 1214–1225. https://doi.org/10.1109/ICDE.2019.00111
[50] Shangyu Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. 2021. Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra. In *SIGMOD*. 1222–1234. https://doi.org/10.1145/3448016.3457317
[51] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.
[52] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. 2019. AutoGraph: Imperative-style Coding with Graph-based Performance. *SysML* (2019). https://proceedings.mlsys.org/book/272.pdf
[53] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577. https://doi.org/10.48550/arXiv.1712.05889
[54] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW*. 383–402. https://dl.gi.de/20.500.12116/2418
[55] Jianmo Ni. 2018. Amazon Product Data - Books. https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/.
[56] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture - UNPRECEDENTED ACCELERATION AT EVERY SCALE. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.
[57] Dan Olteanu. 2020. The Relational Data Borg is Learning. *PVLDB* 13, 12 (2020), 3502–3515. https://doi.org/10.14778/3415478.3415572
[58] Kunle Olukotun. 2021. Keynote: "Let the Data Flow!". In *CIDR*.
[59] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy G. Mattson. 2016. The TileDB Array Data Storage Manager. *PVLDB* 10, 4 (2016), 349–360. https://doi.org/10.14778/3025111.3025117
[60] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees. In *SIGMOD*. 1135–1152. https://doi.org/10.1145/3299869.3300077
[61] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091. https://doi.org/10.14778/2536222.2536233
[62] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*. 858–869. http://dl.acm.org/citation.cfm?id=1164201
[63] Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. 2021. Combiner: Full Attention Transformer with Sparse Computation Cost. In *NeurIPS*. https://proceedings.neurips.cc/paper_files/paper/2021/file/bd4a6d0563e0604510989eb8f9ff71f5-Paper.pdf
[64] Youcef Saad. 1994. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2.
[65] Brennan Saeta. 2018. Training Performance A user's guide to converge faster, TF Dev Summit.
[66] SAP. 2019. SAP HANA Performance Guide for Developers. https://help.sap.com/doc/05b8cb60dfd94c82b86828ee77f7e0d9/2.0.04/en-US/SAP_HANA_Performance_Developer_Guide_en.pdf.
[67] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*. 3–18. https://doi.org/10.1145/2882903.2882939
[68] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD*. 1642–1659. https://doi.org/10.1145/3299869.3324961
[69] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH*. 1058–1062. http://www.isca-speech.org/archive/interspeech_2014/i14_1058.html
[70] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*. 1607–1623.
[71] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *SSDBM*. 1–16. https://doi.org/10.1007/978-3-642-22351-8_1
[72] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, and Jeffrey Clune. 2020. Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data. In *ICML*. 9206–9216. https://doi.org/10.48550/arXiv.1912.07768
[73] Yasuo Tabei, Hiroto Saigo, Yoshihiro Yamanishi, and Simon J. Puglisi. 2016. Scalable Partial Least Squares Regression on Grammar-Compressed Data Matrices. In *SIGKDD*. 1875–1884. https://doi.org/10.1145/2939672.2939864

[74] Teradata. 2021. Teradata Documentation - Table Statements, USING FAST MODE. https://docs.teradata.com/r/76g1CuvvQlYBjb2WPIuk3g/Ls7re6GN7m3Tw_faIYIN2Q.
[75] Ramakrishna Varadarajan, Bibek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. 2014. DBDesigner: A customizable physical design tool for Vertica Analytic Database. *ICDE*, 1084–1095. https://doi.org/10.1109/ICDE.2014.6816725
[76] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems. *PVLDB* 13, 11 (2020), 2662–2675. https://doi.org/10.14778/3565838.3565858
[77] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *NeurIPS*. 7686–7695. https://proceedings.neurips.cc/paper/2018/hash/335d3d1cd7ef05ec77714a215134914c-Abstract.html
[78] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Ce Zhang, and Onur Mutlu. 2019. Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-precision Learning. *PVLDB* 12, 7 (2019), 807–821. https://doi.org/10.14778/3317315.3317322
[79] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394. https://doi.org/10.14778/1687627.1687671
[80] Stephen Wolfram. 2002. *A New Kind of Science - Data Compression.* Wolfram-Media. 1069 pages.
[81] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD*. 2639–2652. https://doi.org/10.1145/3448016.3457566
[82] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning Data Layouts for Big Data Analytics. In *SIGMOD*. 193–208. https://doi.org/10.1145/3318464.3389770
[83] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.
[84] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. 2017. ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning. In *ICML*. 4035–4043. https://proceedings.mlr.press/v70/zhang17e.html
[85] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2017. Trained Ternary Quantization. In *ICLR*. https://openreview.net/forum?id=S1_pAu9xl
[86] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59. https://doi.org/10.1109/ICDE.2006.150

Received April 2022; revised July 2022; accepted August 2022