Morphing-based Compression for Data-centric ML Pipelines

Sebastian Baunsgaard
Technische Universität Berlin & BIFOLD
baunsgaard@tu-berlin.de

Matthias Boehm
Technische Universität Berlin & BIFOLD
matthias.boehm@tu-berlin.de

ABSTRACT

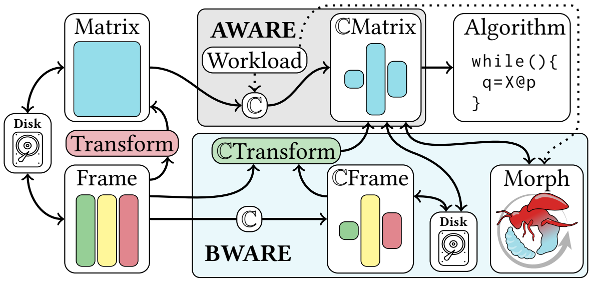
Data-centric ML pipelines extend traditional machine learning (ML) pipelines—of feature transformations and ML model training—by outer loops for data cleaning, augmentation, and feature engineering to create high-quality input data. Existing lossless matrix compression applies lightweight compression schemes to numeric matrices and performs linear algebra operations such as matrix-vector multiplications directly on the compressed representation but struggles to efficiently rediscover structural data redundancy. Compressed operations are effective at fitting data in available memory, reducing I/O across the storage-memory-cache hierarchy, and improving instruction parallelism. The applied data cleaning, augmentation, and feature transformations provide a rich source of information about data characteristics such as distinct items, column sparsity, and column correlations. In this paper, we introduce BWARE—an extension of AWARE for workload-aware lossless matrix compression—that pushes compression through feature transformations and engineering to leverage information about structural transformations. Besides compressed feature transformations, we introduce a novel technique for lightweight morphing of a compressed representation into workload-optimized compressed representations without decompression. BWARE shows substantial end-to-end runtime improvements, reducing the execution time for training data-centric ML pipelines from days to hours.

PVLDB Reference Format:
Sebastian Baunsgaard and Matthias Boehm. Morphing-based Compression for Data-centric ML Pipelines. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at www.github.com/damslab/reproducibility/tree/master/vldb2026-BWARE.

1 INTRODUCTION

Modern machine learning (ML) training comprises more than just selecting and fitting ML algorithms or neural networks and their hyper-parameters. Data-centric ML pipelines extend traditional ML pipelines of feature transformations and model training by pre-processing steps for data validation [40, 94], data cleaning [98], feature engineering [93], and data augmentation [60, 89, 90, 111] to construct high-quality datasets with good coverage of the target domain. These pre-processing techniques can substantially improve model accuracy [60, 98], fairness [93, 102], and robustness [107].



```
while(){
  q=X@p
}
```

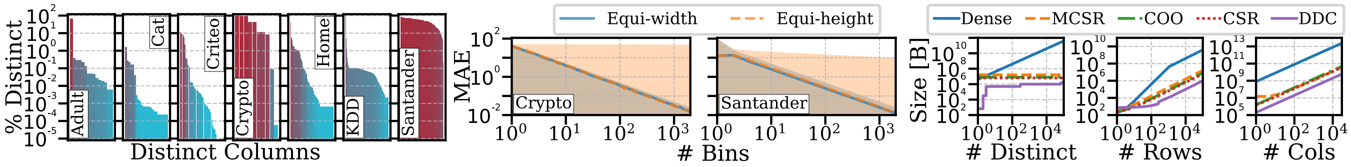
Figure 1: BWARE Framework Overview and Contributions.

Sources of Redundancy: The iterative nature of finding good data-centric ML pipelines causes both operational redundancy (e.g., fully or partially repeated pre-processing steps) [85] as well as data redundancy [13]. Besides natural data redundancy, such as the small cardinality of categorical features and column correlations [35], data-centric ML pipelines create additional redundancy. Examples are the augmentation of data points or features (correlation), as well as systematic transformations such as the imputation of missing values by mean/mode (default values) or MICE [109] (correlation), outlier removal (sparsity/default values), and data cleaning by robust functional dependencies [33] (correlation). While being beneficial for model quality, the iterative selection of such data-centric ML pipelines is a very expensive process. Eliminating unnecessary redundancy through data reorganization is appealing because it can speed up pre-processing and repeated model training, while reorganization overheads can be amortized.

Lossless Matrix Compression: A common approach for exploiting data redundancy without quality degradation is lossless compression. First, sparsity exploitation avoids processing zero values via dedicated data layouts, sparse operators, and even sparsity-exploiting ML algorithms [119]. Common layouts include the compressed sparse rows (CSR), columns (CSC), or coordinate format (COO) [52, 56, 76, 92, 99]. Second, existing compression techniques apply lightweight database compression schemes—such as dictionary encoding, run-length encoding, and offset-list encoding—to numeric matrices and perform linear algebra (LA) operations such as matrix multiplications directly on the compressed representation. Example frameworks are Compressed Linear Algebra (CLA) [34, 35], Tuple-oriented Compression (TOC) [68], Grammar-compressed Matrices (GCM) [37], and AWARE [13] (see Figure 1, top). AWARE creates compressed matrices (ℂ) in a workload-aware manner by (1) extracting a workload summary of the program at compile-time, as well as (2) workload-aware compression and compression-aware recompilation at runtime. Existing work struggles though to efficiently rediscover structural data redundancy in data-centric ML.

A Case for Compressed Pre-processing: Feature transformations encode categorical and numerical features into numerical matrices [84]. This conversion is a rich source of information about

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(a) Relative Number of Distinct Values. **(b) Lossy Quantization Effect on Values.** **(c) Result Sizes of One-Hot/Dummy Coding.**

Figure 2: Three Figures Highlighting Properties of Data Containing Exploitable Compression Potential.

structural data redundancy. For example, one-hot encoding a categorical feature requires determining the dictionary of d distinct items and creating d perfectly correlated binary features. Transformations like binning and feature hashing represent user-defined, lossy decisions which give upper bounds for code word sizes as well. Furthermore, data-centric ML pipelines iteratively evaluate additional features and different transformations. Therefore, we make a case for *pushing compression through feature transformations and feature engineering to the sources* in order to speed-up feature engineering and reduce the overhead of rediscovering data redundancy during matrix compression. Support requires (1) compressing frames in a form amenable to compressed feature transformations, (2) supporting compressed I/O, as well as (3) compressed feature engineering and feature transformations. Since data and workload characteristics of enumerated ML pipelines vary, iterative ML training would benefit from morphing [27, 29, 42] compressed intermediate matrices into workload-optimized layouts [13].

Contributions: In this paper, we introduce BWARE (see Figure 1, bottom) as a holistic, lossless compression framework for data-centric ML pipelines. Our main technical contributions are:

- A lightweight frame compression scheme with dictionary encodings, enabling compressed feature transformations on heterogeneous data (Section 3).
- A novel morphing technique for workload-aware tuning of compressed representations (Section 4).
- Parallel and distributed I/O primitives for compressed blocks without decompression (Section 5).
- An optimizing compiler that injects morphing instructions into linear algebra programs (Section 6).
- An experimental evaluation that studies the impact of compressed I/O, feature engineering and transformations, as well as ML training in data-centric ML pipelines (Section 8).

2 EXPLOITABLE REDUNDANCY

We aim to quantify the potential of exploiting structural and value-based data redundancy. To this end, we first summarize data characteristics of real-world datasets and investigate the potential runtime impact of pushing compression through pre-processing primitives. For notation, $\mathbf{X}$ is an $n \times m$ frame or matrix, where each column $\vec{x}_i$ has a type $\in \{\mathbb{R}, \mathbb{N}, \mathbf{Str}, \mathbf{Hash}\}$, n cells, and d_i distinct values.

2.1 Distinct Values

The number of distinct values d is a classic characteristic to exploit in compression [26, 73], especially in string columns. Dense Dictionary Coding (DDC) constructs a dictionary of d values and encodes the values as integers positions in the dictionary. Figure 2a shows the cardinality ratios d_i/n for datasets characterized in Table 4. Some columns contain less than 0.001% distinct values. Compressed operations that exploit $d \ll n$ can reduce execution time in such cases by 99.999% [13]. Unfortunately, there are also columns with many distinct items, motivating additional encodings.

2.2 Lossy Transformations

Feature engineering can reduce and control the number of distinct vales d via lossy transformations such as binning, feature hashing, or quantization. An example of static quantization is equi-width binning, which scales the input values to discretized bins in the range of min-max with $Q_\Delta(\mathbf{X}) = \hat{\mathbf{X}} = \lfloor \Delta(\mathbf{X} - \mathbf{X}_{min})/(\mathbf{X}_{max} - \mathbf{X}_{min}) \rfloor$. The resulting number of distinct values is $d \leq \Delta$, where Δ is the configured number of bins. Increasing Δ generally improves the accuracy of the approximation of the original data. $\Delta = 256$ is a common configuration, which allows encoding values in UINT8. Figure 2b shows the relative loss of equi-width and equi-height quantization. Equi-height quantization maps input values to buckets by Δ quantiles. The x-axis varies Δ and the y-axis shows the mean absolute error: $\text{MAE}(\mathbf{X}, \hat{\mathbf{X}}) = \sum_1^{nm} |x_i - \hat{x}_i|/nm$. The upper and lower bounds of the blue and orange colored areas are the min/-max absolute errors. The plot shows a linear relationship (log-scale plots) of roughly $\text{MAE}(\mathbf{X}, Q_\Delta(\mathbf{X})) \approx 2 \cdot \text{MAE}(\mathbf{X}, Q_{2\Delta}(\mathbf{X}))$, meaning if Δ doubles, the MAE error is halved. Learned quantization schemes [120, 122] use various techniques to find optimal quantization boundaries (smaller bins for high-frequency value ranges). Learned schemes can improve the MAE using higher Δ or optimize for other goals such as model accuracy, or compression size.

2.3 Non-numerical Data

Categorical values are commonly encoded with one-hot encoding, whereas text is often represented via word embeddings. Feature transformations producing numerical representations through binning, feature hashing, recoding, and one-hot encoding have the potential to compress encoded values. Some are lossy categorical transformations that reduce d. Feature hashing maps values to Δ buckets, and for Natural Language Processing (NLP), one can limit the number of unique words or tokens (d) for encoding via lemmatization [30] and stemming. Figure 2c shows the potential of compressing one-hot-encoded columns using dictionary compression compared to different sparse representations. The three sub-plots systematically vary the number of distinct values, rows, and columns of inputs (with base parameters d = 1,000 distinct values, n = 100K rows, and m = 5 columns). The output shape is $[n, md]$, and sparsity is $1/d$. The y-axis shows the in-memory size in bytes of the encoded output matrix. The dense allocations show worse performance in all cases beyond very few distinct values or rows. Sparse layouts such as CSR [12], COO [12], or Modified CSR (MCSR) [15] yield good compression in all cases. Sparsity exploitation performs exceptionally well when scaling d. However,

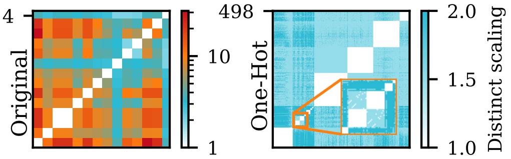


Figure 3: Relative d Increase when Co-coding Features in Adult: Original and One-Hot Encoded Features.

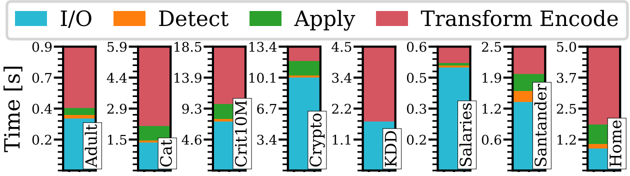


Figure 4: Breakdown of Reading, Detecting and Applying Schemas, and Losslessly Encoding Different Datasets.

DDC [9, 13, 34, 73, 115] allocates the least memory. The dampened size increase for Dense in the middle plot is because d (and thus, the number of one-hot encoded columns) increases only until 1,000.

2.4 Correlation

Column correlation also impacts compressibility. Figure 3 shows the relative increase in the number of distinct tuples when jointly encoding different columns in the Adult dataset (we removed one column with $d > 20k$). The left sub-figure shows the original features, while the right shows the one-hot encoded categorical features. Let $d_{i,j}$ be the number of distinct tuples of the co-coded columns i and j. Then, each cell $c_{i,j}$ shows $c_{i,j} = 2d_{i,j}/(d_i + d_j)$ as the relative increase of distinct tuples if the columns are combined. White ($c_{i,j} = 1$) indicates columns with perfect correlations, which, for instance, is the case for all pairs of one-hot-encoded columns originating from the same column. Ideally, co-coding would first group one-hot encoded features with perfect correlation and subsequently other correlated features. Rediscovering the correlation between many columns after feature transformations is non-trivial and potentially very expensive since each combination of columns has to be analyzed on a sample. A greedy co-coding algorithm requires $O(n^2)$ time to discover these correlated columns, which makes it even more expensive after transformations that increase the number of columns, such as one-hot encoding. The rediscovery is further complicated by ultra-sparse matrices and the existence of sparsity-exploiting compression schemes, where the full co-coding potential is often not analyzed in favor of fast compression. Interestingly, Figure 3 shows a perfect correlation between the original features 3 and 4, while the one-hot encoded version does not perfectly co-code on all pairs of columns (see zoomed-in area). This perfect correlation is only detectable when evaluating larger sub-groups. Therefore, pushing compression through feature transformations has the potential for both runtime reduction and improved compression.

2.5 Pre-processing Time

Figure 4 shows the execution time of pre-processing the different datasets (Table 4). We read CSV files from disk, marked as I/O. In case of unknown data types, schema detection and application aim to specialize generic strings into integer and floating point data types where possible. We detect data types on a sample and apply them during data conversion. As a final step, the heterogeneous frame is transformed into a homogeneous matrix. All these stages can be improved via compression. Reading compressed representations from disk reduces the number of read bytes, frames saved with a schema can skip schema operations, and feature transformations can be done efficiently on compressed formats.

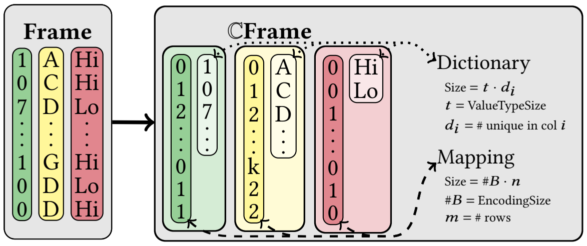


Figure 5: The Compressed Frame Format.

3 COMPRESSED DATA PREPARATION

This section describes BWARE's compressed frame layout, feature transformations to matrices, and feature engineering.

3.1 Compressed Frame Design

Uncompressed frames are tables stored in columnar arrays. Each column can contain a different value type. Figure 5 shows our ℂFrames using a dense dictionary coding (DDC) scheme per column. Each DDC column consists of a mapping array of length n on the left, and a dictionary array of length d_i on the right. The map contains value positions in the dictionary.

Compressed Size: The compressed size of a column depends on d_i, n, and value type size t. The encoding scales according to $n\#B + d_i t$ where $\#B \approx \lceil \log_2(d) \rceil$ bits/value. The mapping supports 0 or 1 bit and 1-, 2-, 3-, or 4 byte encodings with up to 1, 2, 256, 64K, 16M, and 2G distinct values. If $n\#B + d_i t > nt$, where nt is the uncompressed size, we fall back to a pointer to the input column. We also compress boolean columns, which always increases their size, but can be leveraged in feature transformations and engineering.

Type Conversion: Additional type conversion can utilize specialized value types. We detect the value type on a sample of the data and fuse conversion and column compression. In case of casting errors, we re-detect a guaranteed correct value type and convert the column to the newly detected type. We support string, int, character, boolean, hexcode, and float types of different precision. The schema detection and application are critical because our system defaults to reading frames as strings unless a schema is provided on the initial read. For example, a hash encoded as a hex `"bcdef123"` but allocated as a string can be very costly.

Simple Compression: We do not co-code Frame columns because many feature transformations use unique dictionaries for individual columns, and different columns can contain different value types. Subsequent workload-aware morphing (Section 4) of the compressed format then anyway creates the final matrix compression with full support of different encoding schemes and co-coding. The proposed transformation techniques would also work with other dictionary-based compression techniques (e.g., RLE [1], SDC [13], and OLE [34]), which we leave as interesting future work.

Table 1: Transform-Encode Feature Transformation Types.

Name	Dummy	Compressed In & Output
Bin, Hash, Pass, & Recode	✓	✓
Word Embedding		✓

Compression Algorithm: We fuse type detection, type conversion, and DDC compression, for each column $\vec{x}_i$ of the input frame. The process consists of the following:

1. Sample values from $\vec{x}_i$ to detect a candidate type. If sampling is inconclusive, scan the full column to ensure consistent and valid type inference.
2. Allocate a *mapping* array of size n to store an integer ID for each entry in $\vec{x}_i$.
3. Traverse all n entries, building a hashmap assigning each distinct value to a unique integer ID in $[0, d_i)$. All values IDs are stored in the *mapping*.
4. If the total compressed cost $n\#B + d_i t$ exceeds the uncompressed cost nt, abort compression for $\vec{x}_i$. However, type detection and conversion may still reduce memory use.
5. Pack the *mapping* into a compact format based on d_i, reducing the bits per entry (approaching $\#B$).
6. Allocate a dictionary array D of size d_i and fill it using the hashmap's key-value pairs $< k_i, v_i >$ and assign $D_{v_i} = k_i$.

Parallelization: We naïvely parallelize over all input columns because we compress frame columns independently. However, some datasets contain few columns and many rows, and parallelizing only over columns does not fully utilize the available degree of parallelism. Therefore, each column thread further parallelizes the parsing of value types from strings—which can be costly (e.g., String to double [65])—over row segments.

3.2 Compressed Feature Transformations

Transform-encode encodes a heterogeneous frame into a homogeneous matrix by applying dedicated feature transformations (built-in function `transformencode`). This operation produces two outputs: A matrix, and a metadata frame for applying the same transformations to other frames and decoding. We support the transformations shown in Table 1. Other numeric transformations can be subsequently performed in linear algebra (e.g., normalization).

Lossless: We support two lossless transformations: *Pass* returns the same values as the input cast to double, but requires numeric inputs. *Recode* encodes input values into contiguous integers for each unique value encountered (DDC encoding, without the dictionary).

Lossy: Similarly, there are two lossy transformations: *Bin* short for Binning, constructs Δ buckets to encode the values into. The bins use equi-height or equi-width quantization. Equi-height constructs buckets with similar frequency of data points by calculating quantile boundaries. Equi-width extracts the minimum and maximum value and constructs buckets of equal ranges. The values returned from binning are bin IDs. Another technique, *hash*, hashes each value and returns the hashed value modulo the maximum number of buckets k to yield a bin ID with $\hat{\mathbf{X}} = \texttt{hash}(\mathbf{X}) \,\%\, k$.

Dummy Coding: Dummy coding encodes integer values v into one-hot sparse vector representations with a single one set in position v. This transformation is typically applied to integer-encoded features and thus, can be combined with other transformations such as recoding, feature hashing, and binning.

Word Embeddings: Encoding words into semantic-preserving numeric vectors is done via *word embeddings*, which is a sequence of *tokenizing*, *dummy coding*, and matrix multiplication with an embedding matrix. n denotes the size of each embedding vector.

Sequences: Figure 6 shows different transformation sequences. In the following, we abbreviate frame compression as F-ℂF and matrix compression as M-ℂM.

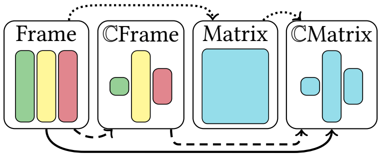


Figure 6: Transform Encode Seq.

Frame to Matrix (F-M-ℂM): The already existing baseline approach is to first transform-encode an uncompressed frame to an uncompressed matrix (F-M). Subsequently, the matrix is compressed (M-ℂM) with existing lossless matrix compression techniques [13, 34, 37, 68]. However, the separate matrix compression has to extract statistics from the intermediate matrix again, many of which are similar to the F-M transformation's statistics.

Frame to Compressed Matrix (F-ℂM): Our novel compressing transformations avoids the double analysis as follows:

- **Recoding:** Uses two passes: (1) construct a hashmap of unique values to continuous IDs, and (2) applying the assigned IDs. Finally, allocate a dictionary using the hashmap keys as values and values as offsets.
  +Dummy: Use an identity matrix of size $d_i{\times}d_i$ as dictionary. This approach works because recoding guarantees incremental IDs from 0 to $d_i - 1$, allowing a simple identity dictionary to represent all possible one-hot vectors.
- **Pass-Through:** Takes a sample if uncompressed and verifies compressibility. If a column is incompressible, return an uncompressed column group. Otherwise, proceed as recode, but use the hashmap keys for the dictionary values.
- **Hashing:** Hashing does not need a hashmap. Instead, we directly allocate a dictionary similar to recode of k values and hash each tuple directly into the mapping. The hashing method may not use all buckets, potentially creating unnecessary entries in the compressed dictionary.
  +Dummy: Use an identity matrix of k rows and columns. Since the number of hash buckets k is known beforehand, the identity dictionary aligns with the fixed range of IDs.
- **Bin:** Calculate the bin ID of each value and put it into the mapping. The dictionary is incrementing integers until Δ.
  +Dummy: Use an identity matrix of Δ rows and columns because binning produces a known number of Δ bin IDs.

Compressed Frame to Compressed Matrix (ℂF-ℂM): Compressed frame inputs offer multiple optimization opportunities. First, we skip constructing hashmaps by directly utilizing the dictionaries of a compressed frame. Second, we reuse the allocated index structures (i.e., the DDC map, and others for other schemes) of the ℂFrame for the ℂMatrix. Compression is usually dominated by creating index structures because the ratio of distinct values is commonly small. Reusing the index structures makes lossless transformations scale according to $O(\sum_{i=1}^{m} d_i)$ rather than $O(nm)$, and with dummy coding $O(1)$. These approaches are, however,

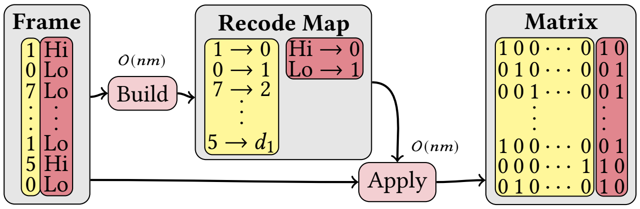


Figure 7: Uncompressed Recode & Dummy-code (One-Hot).

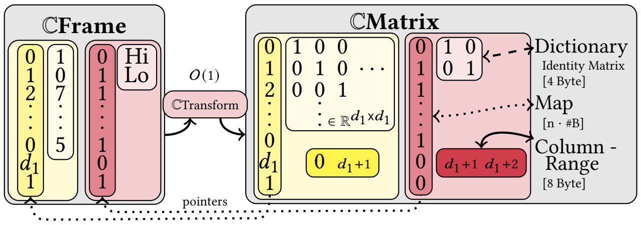


Figure 8: Compressed Recode & Dummy-code (One-Hot).

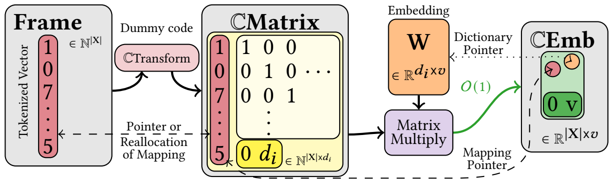


Figure 9: Compressed Linear Algebra Word Embedding.

only applicable if we use lossless transformations because lossy transformations have to reallocate or re-map their index structures.

Example: *[Figure 7](#) shows recoding and dummy coding an uncompressed frame of two columns (left) to an uncompressed matrix (right) with $O(nm)$. Unique values are first mapped and recoded to contiguous integers. The final dummy coding then returns d_i columns for each input column i. Figure 8 shows the equivalent operation in compressed space with time complexity $O(1)$. The dictionary is a virtual identity matrix of size $\mathbb{R}^{d_i \times d_i}$ stored in a single integer. Furthermore, each output column group contains a column range (for later co-coding). The mapping size depends on the number of rows n and d_i. Assuming $n = 1{,}000$, $d_1 = 200$, and $d_2 = 2$ the left mapping uses 1 byte/row and the right uses 1 bit/row. The ℂMatrix then requires* $1{,}032 + 176 = 1{,}208$ *byte plus object and pointer overheads.*

Compressed Word Embedding: Figure 9 shows how we perform a compressed word embedding for a single column input in $O(1)$, only requiring shallow copies of (i.e., pointers to) already allocated intermediates. Since the embedding is a right matrix multiply and the intermediate compressed matrix's dictionary is an identity matrix, the embedding multiplication constructs a new compressed result with a pointer to the full embedding matrix, reused as the dictionary of the dictionary encoding.

Intermediate Sizes: Table 2 shows the size formulas of the different outputs. **F-M** is the uncompressed standard transformation, while **F-ℂM** and **ℂF-ℂM** produce compressed matrices. The one-hot **F-M** size assumes CSR output. Otherwise, dense representations require $8nd_i$ byte. '*constant*' means the compressed input index structure is used directly in the output.

3.3 Compressed Feature Engineering

Compressed feature engineering constructs additional features (e.g., data augmentation) directly on the compressed representation.

Table 2: Transform-Encode Column Size Scaling.

	F-M	**F-ℂM**	**ℂF-ℂM**
Recode & Pass	$8n$	$\#Bn + 8d_i$	*constant*
Bin & Hash	$8n$	$\#Bn + 8\Delta$	$\#Bn + 8\Delta$
With One-Hot / Dummy-Coding			
Recode & Pass	$12n$	$\#Bn$	*constant*
Bin & Hash	$12n$	$\#Bn$	$\#Bn$
Word Embed	$(8v + 12)n$	$\#Bn$	constant

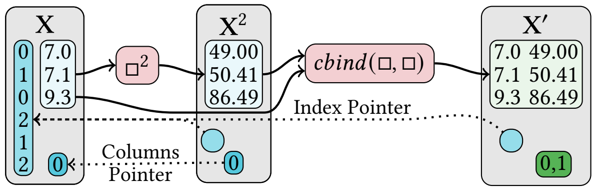


Figure 10: Extending Features in a Co-coded Column Group.

Modified Features: We define feature modifications as element-wise operations that change all feature instances equally. A typical example is normalization such as standard or min-max scaling. These modifications are needed for well-behaved ML training. Similar to prior work [13, 34, 37, 68], we perform element-wise operations in compressed space, with (for most operations) time complexity in the number of distinct items $O(d_i)$ per column group.

Additional Features: An example of concatenating $\mathbf{X}$ with additional features $\mathbf{X}' = \text{cbind}(\mathbf{X}, \mathbf{X}^2)$ are squared features. Such features allow simple linear models to take non-linearities into account, similar to the effect of kernel-based methods [8], though in our case the expanded feature space is explicitly materialized. While such appends normally require allocating the extended matrix, BWARE exploits their perfect correlation. Figure 10 shows the modification on a DDC encoding. First, we perform a scalar power operation, which is a dictionary-only operation, only allocating a new dictionary and maintaining pointers back to the original input mapping. Second, when concatenating columns (*cbind*), we detect that both indexes point to the same mapping, indicating perfect correlation and allowing the direct combination as a co-coded column group. Similar exploitation strategies exist when subsets of columns are modified and appended.

Performance: The cardinality ratio n/d_i defines the potential speedup of compressed feature engineering of individual columns because the new features have perfect correlation with the original features and can share index structures. Exploiting the co-coding removes redundant compression analysis of the augmented matrices. Many compressed operations benefit from extensive co-coding. For example, left matrix multiplication (LMM), with compressed right and uncompressed left inputs, benefits because pre-aggregation is independent of the number of co-coded columns.

4 MORPHING

Our morphing-based compression transforms uncompressed dense, sparse, or compressed matrices into workload-optimized compressed matrices (e.g., after feature transformations but also during compressed I/O). The morphing workflow is shown in Figure 11. For uncompressed inputs, we extract statistics and group columns

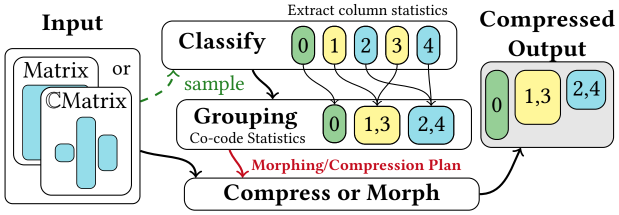


Figure 11: Morphing or Compression Workflow.

Algorithm 1 Morphing DDC Combine.

```
Require: DDC$_{Map}$: $I_1, I_2$, DDC$_{Dict}$: $D_1, D_2$, DDC$_{Cols}$: $C_1, C_2$
  $\mathbf{M} \leftarrow$ HashMap, $I_R \leftarrow$ constructIndex(len($I_1$))
  for $i_1, i_2, i_R$ in $I_1, I_2, I_R$ do $i_R \leftarrow$ M.putIfAbsent($i_1 + i_2 d_1$, M.size())
  $D_R \leftarrow$ constructDictionary(size(M), len($C_1$) + len($C_2$)))
  for $k, v$ in M do $D_R[v] \leftarrow$ combine($D_1$.get($k\%d_1$), $D_2$.get($k/d_1$))
  return DDCColGroup($I_R, D_R$, combine($C_1, C_2$))
```

according to these statistics and the workload. In case of a compressed matrix input, we reuse the statistics of pre-existing co-coded columns and skip unnecessary exploration. The result is a morphing plan containing which columns to merge into what encoding.

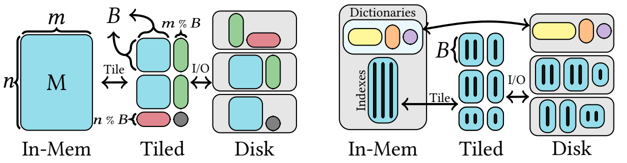
Morphed Combining of Compressed Columns: To avoid decompression, we devise a co-coding algorithm that takes two encoded columns and returns a compressed co-coded column. Algorithm 1 shows the combination of DDC column groups. A naïve combination would produce the Cartesian product $d_i d_j$ of the dictionaries, while our solution materializes only dictionary tuples that co-appear in the index structures $d_{i,j}$ where $d_{i,j} \geq d_i \wedge d_{i,j} \geq d_j$. Instead of populating the combined index with the naïve Cartesian product's index values, we indirectly populate a hashmap to assign the combined index. Each combined dictionary tuple can then be looked up in the hashmap. The asymptotic time complexity of this morphed co-coding is $O(n)$, because typically $d_{i,j} \leq d_i d_j \ll n$.

Morphing a Column Encoding: Combined column groups might not be in the correct encoding per the morphing plan. Therefore, a final phase morphs group encoding types. Since most of our encodings are variations of DDC, the conversion is simple. In general, we try to change encodings while reusing intermediates as much as possible. In practice, changing encodings typically only modifies the index structure while keeping the dictionaries.

Fallback Morphing Execution: Sometimes, the set of column groups selected for co-coding uses heterogeneous encoding schemes, making it hard to have specialized combining algorithms for all. The fallback solution—in case specialized kernels are non-existing for combinations of encodings—is to decompress the selected morphing columns into a temporary matrix followed by a standard compression from scratch. This fallback allocates a potentially expensive uncompressed matrix of $\mathbb{R}^{nxm}$ to combine. The fallback case is often avoided because the `transformencode` currently only uses DDC. We have specialized methods for most permutations of SDC [13], DDC, CONST, EMPTY and Uncompressed column groups, therefore, avoiding the fallback.

5 COMPRESSED I/O

Prior work on lossless compressed linear algebra [13, 34, 37, 67] uses online compression after local or distributed reads, where—at least for local compression—the entire uncompressed input matrix needs to fit in memory. This approach restricts the size of compressible matrices and redundantly compresses the matrix for every program execution. To address these limitations, the BWARE compression framework can read and write compressed blocks and support continuous compression of streams (collections) of blocks.



(a) Uncompressed (b) Compressed

Figure 12: Compressed and Uncompressed Tiled Formats.

Uncompressed Format: Our on-disk uncompressed format is a tiled format allowing distributed reads of collections of pairs of index and matrix blocks. Figure 12a shows this structure. The format excels in reading from local and distributed (HDFS) storage since multiple partitions can be read in parallel [32] via e.g. Spark.

Partitions & Tiles: The tiles are grouped into partitions, each written to a separate file. The number of tiles in each partition is determined by the partition sizes. The partition sizes are tailored to the underlying system disk block sizes to improve disk utilization. We use partition sizes of 128MB in HDFS (default block size) and 16KB in local. Partitions are allowed to grow larger than the minimum size, but preferably multiples of the base disk block sizes.

Compressed Disk Format: To retain efficient distributed processing, we reuse the tiled format for compressed I/O. This design introduces challenges, especially in avoiding decompression. When reading, we need to combine multiple, potentially differently compressed, tiles. When writing, we need to tile the compressed blocks. As shown in Figure 12b, index structures and dictionaries are split into separate partition files. For local writes, dictionaries are written once and combined with the tiled indexes during reads. In distributed settings, each block is compressed independently, storing both index structures and dictionary. The distributed design avoids the need to merge compressed blocks for deduplication but increases compression size due to redundant dictionaries.

Local Reading: When reading a compressed matrix into local memory, we consolidate blocks into a single columnar compression scheme. Consolidation occurs when reading from disk, but also when collecting compressed Spark intermediates. If all blocks in a column use the same compression scheme, only one dictionary is used, and index structures are combined directly. Otherwise, some blocks may require decompression, morphing, or re-compression. Morphing changes the compression of sub-blocks into the consolidated scheme without decompression. Some group encodings can directly combine with others, such as `CONST`, `EMPTY` and `DDC`.

Distributed Reading: To read a compressed matrix in Spark (as a distributed backend of SystemDS), we construct a sequence of lazily evaluated RDD operations that materialize compressed sub-blocks. If no separate dictionary file exists, all tiles must be self-contained with both index structures and dictionaries, and we return a *PairRDD* of indexes and blocks. If a dictionary file exists, we first read index structures into *PairRDDs* of indexes and blocks, and the dictionaries into another *PairRDD* with the same indexes. We then join them to construct fully self-contained compressed blocks. If the dictionaries grow beyond the tile size, we increase the default

Algorithm 2 Serialized DDC Fused Update and Encode.

```
Require: $Matrix$ : X, Scheme: $S$; $C \leftarrow S_{Cols}$: , $M \leftarrow S_{Map}$, $D \leftarrow S_{Dict}$
  $m_s \leftarrow M$.SIZE()                          ▷ HashMap size before updates
  $I_R \leftarrow$ CONSTRUCTINDEX(ROWS(X))           ▷ Allocate output index
  for $r$ in X.ROWS() do
      $t \leftarrow$ EXTRACTTUPLE($r$, X, $C$)       ▷ Extract tuple from row
      $i \leftarrow$ M.PUTIFABSENT($t$, $M$.SIZE())  ▷ Increment size on new
      if $I_R$.NOTVALID($i$) then FAIL               ▷ Check support of value
      else $I_R[r] \leftarrow i$
  if $m_s < M$.SIZE() then $D \leftarrow$ UPDATEDICT($M$, $D$)
  return DDCCOLGROUP($I_R$, $D$, $C$)
```

```
Fx = read($1) ①
Y = read($2)
parfor(t in transformation_specs){ ②
  Mx = transformencode(Fx, t)
  parfor(a in augment_specs){ ③
    Ax = augment(Mx, a)
    print(lmCG(Ax, Y))}} ④
```

Figure 13: Data-Centric ML Pipeline Pseudo-code

block size as a fallback, which happens rarely, since blocks are written using the smallest of compressed, dense, or sparse formats.

Update & Encode: To support large matrices or streaming use cases (e.g., continuous data collection), we apply a compression plan to a stream of incoming matrix blocks. This technique enables dynamic updates to the compression scheme without full analysis, and each scheme can run in parallel. Algorithm 2 shows the DDC encoding variant (one of seven supported encodings). Given a compression scheme (derived from a sample), we first attempt a fused compression that processes the input block in one pass, optimizing for memory bandwidth. We allocate the output index structure, initialized based on the previously seen number of distinct values. The for loop extracts the co-coded value tuples from the matrix and probes the map: new tuples are assigned contiguous IDs, while existing ones reuse their ID. If many distinct tuples appear, the index structure may fail to encode them, causing an abort and fallback to a two-pass algorithm that updates and encodes separately. If no new tuples are encountered, the previous dictionary is reused. Otherwise, the dictionary is updated. This design allows *all* previously encoded blocks to share the latest dictionary for computations.

6 COMPILER AND RUNTIME INTEGRATION

Data-centric ML pipelines transform data through multiple stages from disk over pre-processing and augmentation to ML algorithms. Figure 13 shows a pipeline containing nested loops for finding the optimal pre-processing primitives. The stages comprise reading ①, a loop for different feature transformation specs t ②, a loop for augmentation strategies a ③, and the training of an ML algorithm ④, exemplified with a conjugate-gradient linear regression (lmCG). There is potential to exploit redundancy. However, adding compression at script level would require hand-tuning the individual combinations of transformations, augmentations, and algorithms used. Instead, we propose an optimizing compiler and runtime that dynamically introduces and executes the compression primitives in a given linear algebra program.

Compiler: We decide, at compile time, where to inject compression and morphing instructions. User-defined linear algebra programs, such as Figure 13, are first compiled into a hierarchy of statement blocks (for conditional control flow and function calls) containing directed acyclic graphs (DAG) of high-level operations (HOPs) per last-level statement block. Each HOP is compiled to one or more low-level operators (LOPs). We detect HOPs with morphing potential by considered operations such as read and transformencode, but also operations like rounding ($\lfloor\rceil$, $\lfloor\rfloor$, $\lceil\rceil$) or comparisons (e.g., $\leq$, $\neq$), which produce integer and boolean outputs. For each candidate HOPs, we then construct a workload vector of program-wide affected, data-dependent operations summarizing the workload costs (i.e., number and cost of executed operations) of the respective intermediate. If the workload summary indicates potential for improvement, the HOP is marked for appending a morphing LOP to its compiled sequence of LOPs.

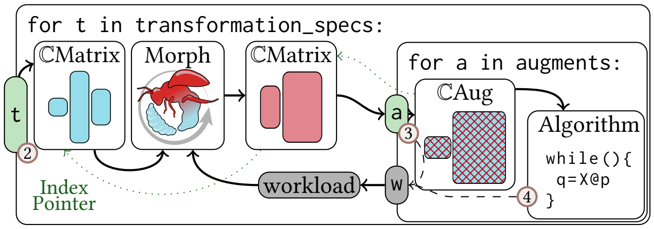


Figure 14: Pipeline with Compiler-introduced Morphing.

Runtime: The runtime morphing has access to the compile-time workload vectors, allowing us to adapt the matrix to the workload. Since morphing supports compressed and uncompressed inputs, we handle unforeseen circumstances (e.g., after conditional control flow and data modification), adjusting the compression while adhering to subsequent workload and data characteristics.

Example: *Figure 14 shows the compiled execution plan for part of the script from Figure 13. In stage ①(not shown) the compiler injects frame compression, depending on the input file, it either compresses an uncompressed input frame or directly reads a ℂFrame. The ℂFrame is transformed into a ℂmatrix, where index structures can be reused, and many transformation costs are $O(1)$. For stage ②, the compiler introduces a morphing instruction to optimize the compressed format according to the workload extracted from the linear algebra program and used operations in stages ③ & ④ (we use DSL-based primitives for augmentation and algorithms). For some operations, the optimizer also introduces morphing into the ML algorithms in ④.*

7 COMPRESSED OPERATIONS

Besides compressed I/O and data preparation, BWARE adds support for new compressed operations, relevant for data-centric ML.

Slicing Ranges: Extracting continuous row ranges from the compressed frames or matrices is efficient by slicing the index structures and retaining pointers to the input dictionaries. We use this for tiling in I/O, Spark broadcasting, and range-based selections.

Selection Matrix Multiply: Selecting a random sample from a matrix can be done via matrix multiplication. If we define a sparse matrix S, with a single 1 in each row and then left matrix multiply it with a matrix **X**, this operation selects rows from **X**. A 1's position in S determines the source row by its column position and the target row by its row position. This multiplication is, for instance, used in the initialization of K-Means where k random rows for each run in K-Means act as cluster starting points. For compressed

Table 3: Compressed Operations, incl. New in BWARE.

$\mathbf{M}$ is a Matrix, $\mathbf{v}_r/\mathbf{v}_c$ are row/col-vectors, s is a scalar, S is a selection matrix, $\square \in \{+,-,/,\cdot,\%,\hat{},|,\&,>,<,==,\leq,\geq,\neq\}$ are operators, $\mathbf{M}^\top$ is transposed, $\mathbb{C}\mathbf{M}$ is compressed, $\mathbf{X}$ is the same input, and @ is matrix multiply.

MatMult	$\mathbb{C}\mathbf{M}@\mathbf{M}$, $\mathbb{C}\mathbf{M}^\top@\mathbf{M}$, $\mathbb{C}\mathbf{M}@\mathbf{M}^\top$, $\mathbb{C}\mathbf{M}^\top@\mathbf{M}^\top$, $\mathbb{C}\mathbf{M}@\mathbb{C}\mathbf{M}$, $\mathbb{C}\mathbf{M}^\top@\mathbb{C}\mathbf{M}$, $\mathbb{C}\mathbf{M}@\mathbb{C}\mathbf{M}^\top$, $\mathbb{C}\mathbf{M}^\top@\mathbb{C}\mathbf{M}^\top$, $\mathbb{C}\mathbf{X}^\top@\mathbb{C}\mathbf{X}$, $\underline{\mathbf{S}@\mathbb{C}\mathbf{M}}$
FusedChain	$\mathbb{C}\mathbf{X}^\top@\mathbb{C}\mathbf{X}@\mathbf{v}_c$, $\underline{\mathbb{C}\mathbf{X}^\top@(\mathbf{v}_r \cdot (\mathbb{C}\mathbf{X}@\mathbf{v}_c))}$, $\underline{\mathbb{C}\mathbf{X}^\top@((\mathbb{C}\mathbf{X}@\mathbf{v}_c) - \mathbf{v}_r)}$
Unary	abs, cos, exp, $\lfloor\rfloor$, $\lfloor\rceil$,$\lceil\rceil$, isNA, isInf, log, ¬, sin, sign, softmax, sqrt, tan, sigmoid,
Binary	$\mathbb{C}\mathbf{M}\square\mathbf{M}$, $\mathbf{M}\square\mathbb{C}\mathbf{M}$, $\mathbb{C}\mathbf{M}\square\mathbf{v}_r$, $\mathbb{C}\mathbf{M}\square\mathbf{v}_c$, and $\mathbb{C}\mathbf{M}\square s$ $\underline{\texttt{removeEmpty}(\mathbb{C}\mathbf{M}, \mathbf{v}_c)}$, replace($\mathbb{C}\mathbf{M}$, s)
Aggregate	sum, $\underline{\text{sd}}$, var, mean, min, max, rowSums, ... , rowMaxs, colSums, ... , colMaxs
Reorg	$\underline{\texttt{rbind}(\mathbb{C}\mathbf{M}, \mathbb{C}\mathbf{M})}$, $\underline{\texttt{cbind}(\mathbb{C}\mathbf{M}, \mathbb{C}\mathbf{M})}$, $\underline{\mathbb{C}\mathbf{M}^\top}$, $\underline{\mathbb{C}\mathbf{M}[s:s, s:s]}$
Transform	$\underline{\texttt{transformencode}(\mathbb{C}\mathbf{F}, s) \rightarrow \mathbb{C}\mathbf{M}}$,$\underline{\texttt{transformencode}(\mathbf{F}, s) \rightarrow \mathbb{C}\mathbf{M}}$

Table 4: Used Datasets and ML Tasks.

Dataset Name	# Row	# Col	Cat	Num	Task
Adult [14]	32,561	15	9	6	Bin
CatInDat [91, 113]	900,000	24	16	8	Bin
Criteo Day 0 [53]	195,841,983	39	25	14	Bin
Crypto [106]	24,236,806	10	1	9	Reg
KDD98 [79]	96,367	481	135	334	Reg
Santander [86]	200,000	201	0	201	Bin
HomeCredit [72]	307,511	121	16	105	Bin
Salaries [10]	397	6	3	3	Bin
AMiner V16 [104]	5,259,857	1,000	1,000	0	Emb

selection multiplication, we use a left compressed matrix multiply that does not pre-aggregate the intermediate matrix, unlike prior work [13, 34, 37, 68]. We use the non-zero values of the left side matrix (guaranteed to be all 1s) to selectively extract compressed row tuples, and decompress them into the output matrix. This solution leverages the index structures of the compression schemes. Extracting larger sub-matrices into compressed intermediates without decompression is interesting future work.

Overview: Table 3 lists the compressed operations supported in BWARE with several new operators—notably, operators for mini-batch algorithms, pre-processing, and writing such as slicing and selection—compared to AWARE [13]. We support most linear algebra operations that are frequently executed on large matrices.

8 EXPERIMENTS

Our experiments study various properties of workload-aware compression. We start with the sizes and compression speed of frames. Then, we compare lossless and lossy approaches to feature transformations. We show how our solution scales by evaluating polynomial feature engineering and highlight a word embedding NLP example with a fully connected layer. Further experiments show end-to-end ML algorithms using both lossless and lossy feature engineering. We also evaluate end-to-end data-centric ML pipelines that iterates through multiple feature transformations. Finally, we compare the transformation performance to other data and ML systems.

8.1 Experimental Setting

HW/SW Setup: All experiments are conducted on a server with two Intel Xeon Gold 6338 2.0-3.2 GHz (64 cores, 128 threads) and 1 TB 3200 MHz DDR4 RAM (peak performance is 6.55 TFLOP/s), 16× SATA SSDs in RAID 0 for the datasets, and an Intel Optane SSD DC P5800X for the programs, scripts, and local evictions (if live variables exceed the buffer pool size). Our software stack comprises Java 17.0.11, SystemDS [15, 16], Hadoop 3.3.6, and Spark 3.5.0.

Baselines: As baselines, we compare BWARE with SystemDS uncompressed I/O and operations (ULA), AWARE [13] for lossless matrix compression, as well as ML systems (i.e., TensorFlow 2.15 [3] and SKLearn 1.4.1 [21, 82]), data management systems (i.e., Pandas 2.1 [105] and Polars 0.20 [112]), and generic compression systems (i.e., ZStd 1.5.5-4 [36] and Snappy 1.1.10.3 [39]).

Datasets: Table 4 summarizes our used datasets, each having different data types and feature transformation requirements. Cat and Num refer to the number of categorical and numerical features, respectively. The tasks are split into regression, binary classification, and word embedding tasks. Adult [14] (also called Census) is a sample from the person records in a 1990 U.S. census. CatInDat [91, 113] (Cat) is a synthetic dataset from Kaggle that contains categorical features for predicting cat ownership, where we combined the two competition datasets. Criteo [53] is a dataset of millions of display advertisements for predicting which ads were clicked. Criteo10M is the first 10 million rows from Criteo. Crypto [106] is a Google competition dataset for time series forecasting of crypto-currencies. KDD98 [79] is a knowledge discovery competition dataset from 1998. Salaries [10] is a small dataset containing the salaries of professors in a U.S. college. Santander [86] is another Kaggle competition to predict customer transactions. HomeCredit [72] predicts how likely each applicant is to repay a loan. Finally, AMiner V16 [104] contains abstracts, where we removed non-English abstracts, equations, symbols, and limited the abstracts to 1000 words.

8.2 Frame Compression and I/O

Figure 15 shows the compressed frames and I/O performance.

In-Memory Size: Figure 15's top row shows three different measures of the in-memory compressed frame. First, *String* represents a frame with the default generic string values without exploiting the value types of the columns. Second, *Detect* automatically detects the value types. *Detect* achieves in-memory size reductions from 1.5x to 18x across the datasets compared to *String*. However, BWARE's compressed frame reduces the size by 19x to 65x. Comparing BWARE with *Detect* shows additional improvements of 1.09x to 43x. A low ratio relative to *Detect* occurs in cases with continuous values and high cardinality, such as Salaries or Crypto. We observe that BWARE requires less memory and guarantees (except for boolean data) smaller or equal sizes to *Detect.* Interestingly, BWARE reduces allocated size even for the tiny Salaries dataset.

On-Disk Size: The second row in Figure 15 shows the size of the datasets on disk. The first column in each figure shows the original size of CSV files. The second column compares our serialized detected frame saved in HDFS sequence files, with tiles of 16K rows. We see that there can be an overhead in storing the tiles. The worst case is KDD98 going from 112MB CSV to 171MB, a 1.5x increase. If HDFS's block-wise compression is enabled, then using Snappy improves KDD98's binary files to 63MB, while ZStd is better with 36MB. Enabling our compressed writers in BWARE, we get 45MB, and ZStd's nested compression 24MB, an on-disk reduction of 4.6x. In conclusion, BWARE performs almost equally to other compression frameworks for individual block compression, and we can recursively stack compression for improvements.

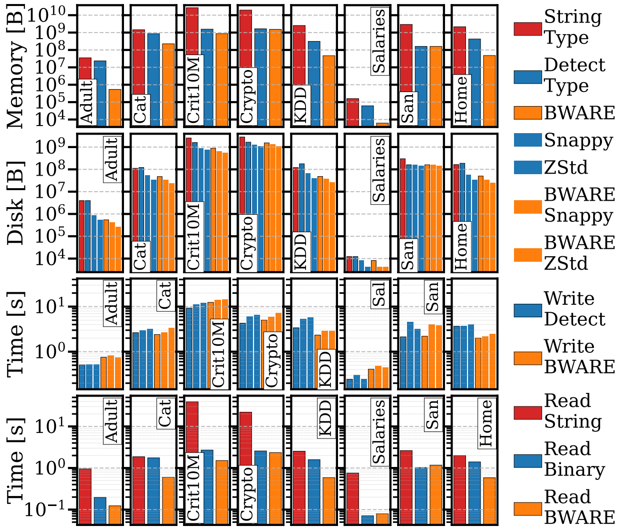


Figure 15: Frame Compression.

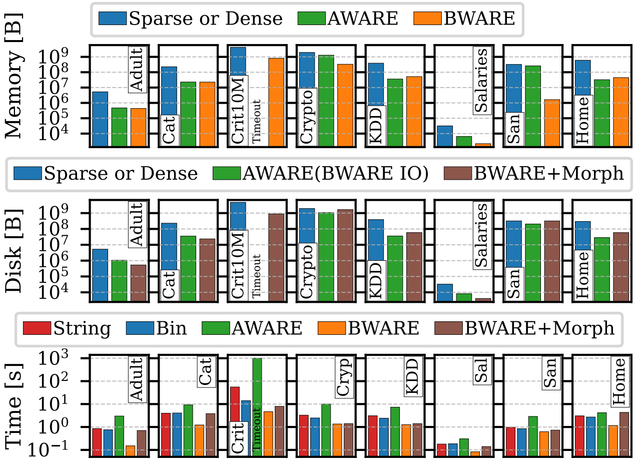


Figure 16: Lossless Transform-Encode.

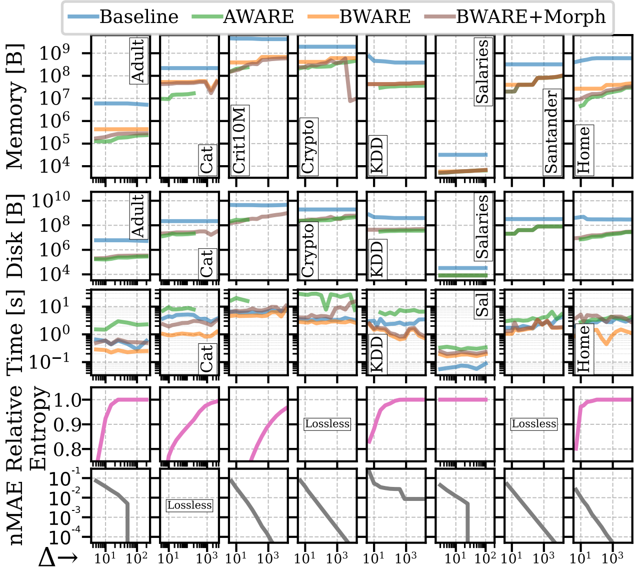


Figure 17: Lossy Transform-Encode Size and Time.

Write/Read Time: The third row in Figure 15 shows the time to write the different formats. The writing time includes schema detection, application, and compression if applicable. BWARE's compression has a comparable overhead to other compressors. The last row in Figure 15 shows the reading performance from CSV, uncompressed, and compressed binary files. We observe that reading text formats should be avoided, but sometimes it is competitive with our binary format (e.g., in Cat). The BWARE reader performs similarly or better than the uncompressed binary reader, even in incompressible cases like Crypto and Santander. The exception is the tiny Salaries dataset, where the binary reader is the fastest.

8.3 Compressed Feature Transformations

Next, we evaluate our compressed feature transformations.

Lossless Encoding: We one-hot-encode all categorical, and pass-through numeric features in a lossless encoding. With this scheme, we copy over values of Crypto and Salaries because all values are numeric, while most columns in Cat and Criteo are one-hot encoded. Figure 16 shows the performance. The rows include (1) the matrix's size in memory, (2) the saved size on disk, (3) the execution time of `transformencode` plus compression or morphing. AWARE and BWARE use less memory than a default sparse or dense matrix, even in the incompressible Crypto and Santander. AWARE's total memory allocation size is sometimes smaller than BWARE. However, BWARE reuses the index structure from the compressed frame arrays in case of 1-to-1 mappings. Furthermore, BWARE's on-disk representation is slightly worse than AWARE (which we extended to use BWARE's I/O operations).

Lossless Time: The last row of Figure 16 shows the transform-encode time. String refers to string inputs, while Binary (F-M) uses detected types. AWARE (F-M-ℂM) encodes detected types followed by compression from scratch, and BWARE (F-ℂM) uses compressed feature transformations. BWARE is faster than the other solutions. AWARE's compression of Criteo shows what happens when the rediscovering of column correlations dominates due to compression after feature transformations (see Figure 3). Criteo is encoded into millions of perfectly correlated columns because of the one-hot encoding. AWARE tries to discover the correlation and starts co-coding. However, due to millions of columns with perfect correlation each co-coding candidate takes time to verify, it takes very long and thus, we timeout the runs at 1,000 seconds.

Lossy Encoding: Figure 17 shows results with different Δ for binning or categorical hash buckets on the x-axis. Note that the information loss is equivalent in all cases. The loss is measured by normalized Mean Average Error (nMAE) and relative entropy = $H(\hat{\mathbf{X}})/H(\mathbf{X})$ where $H(\mathbf{X})$ is Shannon's entropy. BWARE without morphing uses the DDC compression of the compressed `transformencode`, while BWARE with morphing additionally morphs the compression scheme after the transformation. AWARE, BWARE, and BWARE+Morphing use less space than the uncompressed baseline. AWARE returns better-compressed results than BWARE because it has a larger exploration space, while BWARE is more optimized for speed and reuse. When writing to disk, we always use morphing to improve the organization. We observe that the morphed representation is close to AWARE's saved format. The results show that BWARE is faster across all datasets while being

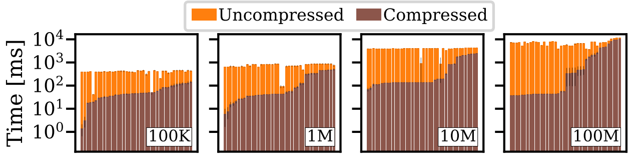


Figure 18: Compressed Input Frames.

on-par for Santander. BWARE yields improvements of 2x to 20x over AWARE and 1x to 5x over the baseline SystemDS.

Encoding Time Breakdown: So far, we used uncompressed frame inputs. If the input frame's columns are compressed, the asymptotic runtime changes to constant for some transformations. Figure 18 shows the parallel lossless transform-encode time of individual columns of Criteo for different numbers of rows with columns sorted by compressed execution time. Some columns in Criteo are compressed, while others have many distinct values and, therefore, are uncompressed. The constant encoding time can be seen in the plateau of the first 50% of the columns. The constant groups take $\approx$ 40ms, except for 10M where it consistently is $\approx 100ms$. The following 25% have to change their compressed index structures, and the final 25% are uncompressed. The two fast columns in uncompressed are boolean columns. Since our hardware setup has a high degree of parallelism, the total encoding time is equal to the tallest bar, while a single-threaded execution is equal to the integral of the colored areas. Therefore, the end-to-end difference between F-ℂM and ℂF-ℂM is small if ℂF contains incompressible columns and there are fewer columns than cores.

8.4 Compressed Word Embeddings

Figure 19 shows the performance of encoding already tokenized abstracts from DBLP [104], padded with zeros to a maximum of 1,000 tokens. All plots show the total execution time of 10 repetitions of encoding word embeddings with word2vec [71]. The first row contains word embedding only, while the second row adds a fully-connected layer with ReLu activation on the embedded outputs. The columns show increasing numbers of unique tokens (d), starting at $d = 1K$ and increasing to $100K$. The x-axis is the number of abstracts (a), encoded, while the y-axis is execution time. We observe that ULA is slower at embedding but as fast as TensorFlow once the network layer is added. ULA catches up because of efficient sparse linear algebra not leveraged in TensorFlow. TensorFlow and ULA are not affected by increasing d, while BWARE is. BWARE shows the best performance in all cases in embedding time and scales further than the other implementations. When adding the neural network layer, the performance is slower in cases where $a < d$. However, once $a > d$, BWARE asymptotically and empirically outperforms all the other implementations.

8.5 ML Algorithm Performance

To quantify BWARE's impact on the end-to-end performance of traditional ML pipelines, we evaluate a conjugate gradient linear regression model (lmCG) training with different lossless and lossy feature transformations as well as feature engineering pipelines.

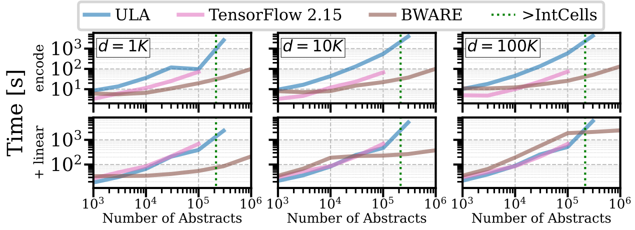


Figure 19: Compressed Word Embedding.

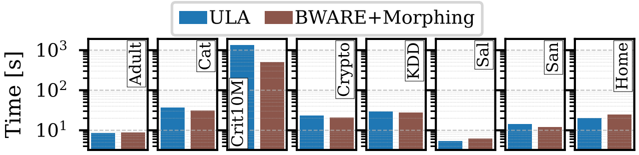


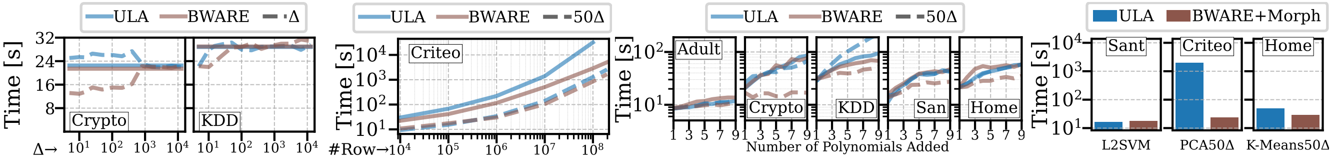
Figure 20: LM Conjugate Gradient Baseline.

Lossless: Figure 20 shows the performance of the lmCG training algorithm. The max #iterations is set to min(m, 1000). The algorithm is sparse-safe, allowing the use of sparse linear algebra. We observe a small slowdown using BWARE of 22% in KDD and 25% in Home. However, Criteo (with 10 million rows) improves by 2x from 1,368 to 681 seconds. Incompressible cases incur minor overhead, but for Crypto and Santander, execution time remains almost unchanged as compression falls back to uncompressed formats. We do not show model accuracies because the results of methods are equivalent. However, as an example, the method scores 78.9 AUC for Cat on Kaggle, while the top score is 80% [91].

Lossy: Figure 21a shows the results when controlling the number of distinct values Δ through lossy feature transformations. The solid lines are the lossless baselines, and dashed lines vary Δ. The Crypto dataset is almost purely numeric and a dense dataset. Due to the large number of rows and many distinct values, we expected and observe benefit from reducing the number of unique values. BWARE is able to exploit the reduced number of unique values, with a increase in runtimes when Δ increases. However, there are also cases that do not benefit, such as KDD with few rows, where only extremely low values of Δ yield performance improvements. Lower Δ generally makes models fit worse, but not always, and sometimes lower Δ can have a positive regularization effect that gives better accuracies. For KDD, the break-even point of lossy and lossless accuracies is $\Delta = 800$. Models are generally able to fit just as well, and sometimes better, on some lossy inputs using only quantization. The results indicate that Δ has a positive impact on runtime with an unknown negative or positive impact on accuracy. Hence, one should perform automated feature engineering.

Scaling: Figure 21b shows the scalability of BWARE in terms of performance on increasingly larger subsets of the Criteo dataset. We observe a starting 2x performance improvement in Figure 20 at 10^4 rows. The improvement increases in all cases until 10^9 rows. BWARE is a substantial 11x faster at 10^8 rows, improving from 31,792s to 2,880s. Utilizing lossy encodings on Criteo, ULA performs better for smaller sizes, but BWARE shows improved performance at scale, with an increasing gap for more rows.

Polynomial Regression: Figure 21c shows the results for regression with polynomial features. BWARE handles these features with moderate overhead and occasional gains. The best case is the Crypto dataset, where the polynomial features do not affect the execution time when combined with lossy feature transformations. In contrast, on the Home dataset with lossless transformations,



(a) lmCG w/ Increasing Δ.

(b) lmCG w/ Increasing n.

(c) Polynomial Regression.

(d) Various ML Algorithms.

Figure 21: Combined Experiments Showcasing Various Properties of BWARE.

Table 5: Pipeline LM: 8 Transform Encode & 8 Polynomials.

Dataset	Measure	ULA	AWARE	BWARE
KDD98	Time	654s	452s	251s
	Instructions	$110 \cdot 10^{12}$	$100 \cdot 10^{12}$	$44 \cdot 10^{12}$
	Instructions per Cycle	0.94	2.59	2.68
	L1-dcache-miss	$7{,}792 \cdot 10^9$	$1{,}740 \cdot 10^9$	$786 \cdot 10^9$
	Compress/Morph	—	148s	21.9s
	Transform-Encode	74.1s	59.9s	7.95s
	Energy [74]	338kJ	185kJ	92kJ
Home	Time	431s	266s	160s
Adult	Time	19.9s	22.1s	16.6s
Santander	Time	489s	376s	374s
Cat	Time	467s	170s	63s

Table 6: 8 Transforms & Polynomials + MICE, and Winsorize

Dataset	Measure	ULA	BWARE
KDD98	Time	919.6s	255.3s
	Compression & Morphing	—	24.9s
	Transform-Encode	37.7s	5.4s
	Instructions	$103 \cdot 10^{12}$	$45 \cdot 10^{12}$
	Instructions per Cycle	0.82	2.68
	L1 Cache Misses	$8{,}276 \cdot 10^9$	$830 \cdot 10^9$

performance drops at low polynomial degrees due to a few incompressible columns (see Figure 2a), whose compression attempts do not amortize. However, lossy transformed columns perform well.

Other Algorithms: AWARE already studied the impact of workload-aware compression on multiple linear-algebra-based ML algorithms, which we inherit for BWARE. Figure 21d shows the performance of additional algorithms. First, BWARE shows equal performance to ULA for L2SVM on Santander. PCA on Criteo with a lossy transformation shows an 83x improvement in execution time. This relative improvement in performance can be arbitrarily large depending on Δ because PCA is asymptotically faster in compressed space. Finally, BWARE shows a solid 2x improvement for K-means on Home using a lossy transformation.

8.6 Data-centric ML Pipeline

Furthermore, we study the execution time and characteristics of end-to-end, data-centric ML pipelines.

Feature Engineering: Table 5 shows the results for an ML pipeline similar to Figures 13 and 14. This pipeline performs a grid search of hyper-parameters with eight Δ ranging from 5 to 480 and eight polynomials from 1 to 8, with two outer loops: for transformencode and polynomial feature construction. The top half of the table shows performance numbers for the KDD dataset. AWARE is 1.45x faster than ULA, and BWARE further improves by 1.8x. BWARE is the fastest because it reuses intermediate compressed representations through the feature transformations, and AWARE rediscovers correlated columns. Here, AWARE uses BWARE's handling of polynomial features. AWARE and BWARE also show better cache locality than ULA, which decreases L1 cache misses by an order of magnitude, and in turn increases instructions performed per CPU cycle. Furthermore, BWARE improves energy consumption due to more efficient data types and reduced cache misses, because data access is a major energy consumer [45]. The bottom half of the table shows the results of the same pipeline on Home, Adult, Santander, and Cat. BWARE is the fastest in all cases. AWARE also does well, except a small overhead on Adult where the compression overhead cannot be amortized.

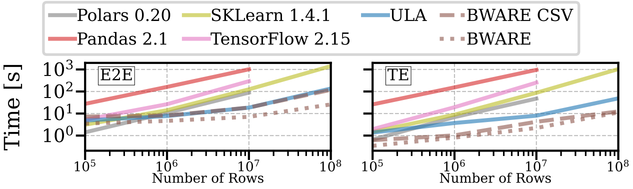


Figure 22: Performance Comparison with Other Systems.

Data Cleaning: Table 6 shows results for a pipeline extended with additional pre-processing steps between feature transformations and polynomial expansions. Specifically, we apply missing value imputation using MICE [109] and perform outlier removal through winsorization [108]. These additional steps increase the overall workload, resulting in better hardware utilization and further widening the performance gap between BWARE and ULA. In this setting, BWARE achieves 3.6x faster processing on KDD.

8.7 Comparisons with Other Systems

Finally, Figure 22 compares the performance of transformencode on Criteo with other systems. The left plot (E2E) shows end-to-end runtimes with CSV parsing, and the right (TE) only transformencode. ULA and BWARE have high startup times in E2E, and JIT compilation benefits in Java are limited for small inputs. However, for larger sizes, ULA (with UPLIFT [84] for feature transformations) and BWARE outperform the other systems. Polars is the only system with a dense matrix result but with UINT8 encoded columns. All other systems use sparse transformations to run until 10^7 rows. At 10^7 rows, BWARE is 4.7x (E2E) and 11.4x (TE) faster than Polars. BWARE's E2E runtimes are equal to ULA when reading CSV, which is still good because it yields a compressed output for subsequent operations (F-ℂM). We also included a dotted line for BWARE reading a compressed frame from disk for compressed encoding (ℂF-ℂM). For more than 2^{32} cells (max 4B integer), many of the systems crash. SK-learn can scale further, but BWARE is 11.9x (E2E) and 76.2x (TE) faster than SK-learn at 10^8 rows.

9 RELATED WORK

BWARE has connections with multiple related fields, including database compression, matrix compression, and specific techniques for workload-awareness and data reorganization.

Database Frame Compression: Tabular data in databases are commonly compressed with schemes exploiting homogeneous column types [2, 100]. Data systems often employ variations of five common lightweight encodings [26, 28]: Frame-of-reference (FOR) [124], delta encoding (DELTA) [124], dictionary encoding (DDC) [9, 73, 115], run-length encoding (RLE) [1] and null suppression (NS) [66, 115]. General-purpose, heavy-weight compressors are applied to compress any data modality. Examples include Snappy [39] and Zstd [36]. Most systems support storing data in compressed formats that can combine multiple techniques. BtrBlocks [62] is a recent work showcasing the effectiveness of nesting compression techniques with highly efficient SIMD decompression. Other examples include Parquet [23], HDF5 [41], and SciDB [101] for storage, as well as Arrow [24, 25] for data transfer. There are also dedicated compression techniques for specific value types such as strings in Pattern-Based Compression (PBC) [123] and FSST [19]. Fine-grained methods, such as white-box compression [38], share some commonalities but are orthogonal. In BWARE, we read and write compressed data blocks—similarly to existing work [63]—but process feature transformations and ML pipelines directly on the compressed formats without decompression.

Lossless Matrix Compression: Compression of numeric data has a long history as well. The most studied type of compression is integer-based compression [26, 28, 62, 66] while floating point data with exponent and mantissa pose some difficulties. Example techniques for floating point data include XOR [88] compression in Gorilla [83], advancements in Chimp [69], and more recently ALP [5]. Sparsity-exploiting compression had already in 1990 full software support with SparseKit [92] using specialized data structures to exploit non-zero values (CSR, CSC, and COO). Sparsity exploitation is now commonly supported in most linear algebra frameworks such as IntelMKL (now behind OneAPI) [52] and CuSparse [76], but is still in active research [56, 99, 116]. UniSparse [70] is a recent example of an MLIR-based [64] sparse tensor system with a compiler optimizing and selecting various sparse formats by—similar to BWARE—decoupling the logical representations from the given user programs. Zhang et al. explores generating code that—similar to our morphing—automatically converts intermediates to sparse tensors [121]. WACO [117] is another recent sparsity-exploiting framework, that optimizes the sparse formats based on operation and data characteristics. More general redundancy exploitation through compression was achieved by Compressed Linear Algebra (CLA) [34, 35], Tuple-oriented Compression (TOC) [68], and AWARE [13], which took inspiration from sparse matrix compression [55, 59]. In contrast to existing work, we push compression through entire data-centric ML pipelines including feature transformations and engineering without recompressing the data.

Lossy Matrix Compression: Mainstream ML systems mostly rely on homogeneous lossy compression partially because it retains regular dense data access. There are three main approaches. First, quantization uniformly encodes all floating point values in fixed or reduced precision [50]. Specialized data types such as Google's bFloat16 [20, 54], Intel's Flexpoint [58] and NVIDIA's TF32 [75] are also very effective. Some solutions use multiple precision levels for different operators [114, 122], even going as far as 1 bit data exchange [46, 96]. Second, dimensionality reduction such as auto encoders [51], PCA [81], or t-SNE [44] are also widely used. Third, sampling or coresets [6] allow training with fewer mini batches [103] or random samples for each batch [78]. Our approach combines user-defined lossy feature transformations with system-level lossless compression to avoid trust concerns in result validity.

Workload-aware Compression: The online, workload-aware compression from AWARE [13] adapts the compressed layout based on workload characteristics of a linear algebra program. Others similarly adapt the data organization based on sparsity [7]. Several systems also combine cost modeling of compressed size and query performance [18, 22, 29, 57, 110], but many of these techniques rely on an offline compression for selection and adapting to workloads [17] or workload traces [80]. In contrast, BWARE performs workload-aware frame and matrix compression in an online manner during runtime before and after feature transformations.

Data Reorganization: Our morphing technique is primarily a data reorganization strategy. Prior work, like database cracking by Idreos et al. [43, 47–49, 87], also dynamically reorganizes data based on query workload. Other work dynamically chooses: (1) the physical design of storing data [11], (2) where to place tuples on distributed servers (e.g. Clay [97]), and (3) online deduplication of stored blocks [118]. MorphStore [26, 42] proposes a morphing wrapper, enabling on-the-fly recompression of intermediate results with lightweight compression schemes for relational algebra. Unlike existing work, we perform workload-aware reorganization of matrices for data-centric ML pipelines.

Compressed Operations: There exist multiple other works for performing operations directly on compressed (matrix) formats. Factorized learning [56, 61, 77, 95] pushes ML workloads through joins, avoiding the materialization of denormalized tables. Grammar-based compressed operations [4, 31, 37] also show good performance, specifically the CSRV representation [37]. CLA [34, 35] supports multiple operations in compressed space, while AWARE [13] extended the operation support to full matrix multiplications. BWARE further extends the operations to feature transformations.

10 CONCLUSIONS

We introduced BWARE as a holistic, lossless compression framework for data-centric ML pipelines, which is fully integrated in SystemDS [15, 16]. In this context, we push compression through feature transformations and feature engineering into the sources. We draw two main conclusions. First, this compression strategy is able to yield substantial runtime improvements because of repeated feature transformations and ML model training. Second, compressed feature transformations preserve information about structural redundancy, achieving improved compression ratios and thus, better data locality for operations. Interesting future work includes support for more feature transformations (e.g., image augmentation) and specialized, heterogeneous hardware accelerators.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Research, Technology and Space (under grant BIFOLD25B).

REFERENCES

[1] Daniel Abadi et al. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. https://doi.org/10.1145/1142473.1142548

[2] Daniel Abadi et al. 2013. *The Design and Implementation of Modern Column-Oriented Database Systems*. https://dl.acm.org/doi/10.5555/2602024

[3] Martín Abadi, Ashish Agarwal, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[4] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. 2020. Impossibility Results for Grammar-Compressed Linear Algebra. In *NeurIPS*, Vol. 33. 8810–8823. https://proceedings.neurips.cc/paper_files/paper/2020/file/645e6bfdd05d1a69c5e47b20f0a91d46-Paper.pdf

[5] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *PACMMOD* 1, 4, Article 230 (2023), 26 pages. https://doi.org/10.1145/3626717

[6] Pankaj K. Agarwal, Sariel Har-Sariel, et al. 2007. Geometric Approximation via Coresets. https://api.semanticscholar.org/CorpusID:13812735

[7] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *PLDI*. https://doi.org/10.1145/3519939.3523442

[8] Mark A. Aizerman, E. M. Braverman, and L. I. Rozonoer. 1964. Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning. In *Automation and Remote Control*, Vol. 25. 821–837. https://cs.uwaterloo.ca/~y328yu/classics/kernel.pdf

[9] G. Antoshenkov, D. Lomet, and J. Murray. 1996. Order preserving string compression. In *ICDE*. 655–663. https://doi.org/10.1109/ICDE.1996.492216

[10] Vincent Arel-Bundock. 2023. *Rdatasets: A collection of datasets originally distributed in various R packages*. https://vincentarelbundock.github.io/Rdatasets R package version 1.0.0 https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/carData/Salaries.csv.

[11] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. 583–598. https://doi.org/10.1145/2882903.2915231

[12] M.A. Bassiouni. 1985. Data Compression in Scientific and Statistical Databases. In *IEEE Transactions on Software Engineering*, Vol. SE-11. 1047–1058. https://doi.org/10.1109/TSE.1985.231852

[13] Sebastian Baunsgaard and Matthias Boehm. 2023. AWARE: Workload-aware, Redundancy-exploiting Linear Algebra. In *PACMOD*, Vol. 1. Article 2. https://doi.org/10.1145/3588682

[14] Barry Becker and Ronny Kohavi. 1996. Adult. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20.

[15] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. In *PVLDB*, Vol. 9. https://doi.org/10.14778/3007263.3007279

[16] Matthias Boehm et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*. https://api.semanticscholar.org/CorpusID:202230751

[17] Martin Boissier. 2021. Robust and budget-constrained encoding configurations for in-memory database systems. In *PVLDB*, Vol. 15. 780–793. https://doi.org/10.14778/3503585.3503588

[18] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*. https://api.semanticscholar.org/CorpusID:81989532

[19] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. In *PVLDB*. https://doi.org/10.14778/3407790.3407851

[20] Google Brain. 2024. bfloat16. https://cloud.google.com/tpu/docs/bfloat16

[21] Lars Buitinck, Gilles Louppe, Mathieu Blondel, et al. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[22] Lujing Cen, Andreas Kipf, et al. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *aiDM*. 4. https://doi.org/10.1145/3464509.3464885

[23] Apache Parquet com. 2023. Apache Parquet: column-oriented data file format designed for efficient data storage and retrieval. https://parquet.apache.org/

[24] Apache Arrow community. 2023. Apache Arrow: A cross-language development platform for in-memory analytics. https://arrow.apache.org/

[25] Apache Arrow community. 2023. Apache Arrow: Dictionary Encoding. https://arrow.apache.org/docs/java/vector.html#dictionary-encoding

[26] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. https://doi.org/10.5441/002%2Fedbt.2017.08

[27] Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2015. Direct Transformation Techniques for Compressed Data: General Approach and Application Scenarios. In *Advances in Databases and Information Systems*, Morzy Tadeusz, Patrick Valduriez, and Ladjel Bellatreche (Eds.). Springer International Publishing, Cham, 151–165.

[28] 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. In *TODS*, Vol. 44. Article 9, 46 pages. https://doi.org/10.1145/3323991

[29] 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*, Vol. 13. 2396–2410. https://doi.org/10.14778/3407790.3407833

[30] J. L. Dawson. 1974. Suffic Removal for Word Conflation. In *ALLC Bulletin*, Vol. 2. 33–46. https://sigir.org/files/museum/pub-21/98.pdf

[31] Rajat De and Dominik Kempa. 2024. Grammar Boosting: A New Technique for Proving Lower Bounds for Computation over Compressed Data. In *SODA*. 3376–3392. https://doi.org/10.1137/1.9781611977912.121

[32] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (jan 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[33] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR*. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf

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

[35] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2018. Compressed Linear Algebra for Large-Scale Machine Learning. In *VLDB J.*, Vol. 27. 719–744. https://doi.org/10.1007/s00778-017-0478-1

[36] Facebook. 2023. Zstandard. https://facebook.github.io/zstd/.

[37] Paolo Ferragina, Giovanni Manzini, Travis Gagie, Dominik Köppl, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. 2022. Improving matrix-vector multiplication via lossless grammar-compressed matrices. In *PVLDB*, Vol. 15. 2175–2187. https://doi.org/10.14778/3547305.3547321

[38] Bogdan Vladimir Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *CIDR*. https://api.semanticscholar.org/CorpusID:210706292

[39] GOOGLE. 2023. Snappy: A compression/decompression library Version 1.1.10.3. https://google.github.io/snappy/.

[40] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *PACMOD* 1, 2 (2023), 128:1–128:26. https://doi.org/10.1145/3589273

[41] The HDF Group. 2023. The HDF5 Library and File Format. https://www.hdfgroup.org/solutions/hdf5/.

[42] Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. 2019. MorphStore - In-Memory Query Processing Based on Morphing Compressed Intermediates LIVE. In *SIGMOD*. 1917–1920. https://doi.org/10.1145/3299869.3320234

[43] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. 2012. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*. https://doi.org/10.14778/2168651.2168652

[44] Geoffrey E Hinton and Sam Roweis. 2002. Stochastic Neighbor Embedding. In *NeurIPS*. https://proceedings.neurips.cc/paper_files/paper/2002/file/6150ccc6069bea6b5716254057a194ef-Paper.pdf

[45] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *ISSCC*. https://doi.org/10.1109/ISSCC.2014.6757323

[46] Itay Hubara et al. 2018. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. In *JMLR*, Vol. 18. 1–30. http://jmlr.org/papers/v18/16-456.html

[47] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*. 68–78. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf

[48] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Updating a cracked database. In *SIGMOD*. 413–424. https://doi.org/10.1145/1247480.1247527

[49] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*. https://doi.org/10.14778/2002938.2002944

[50] IEEE. 2008. Standard for Floating-Point Arithmetic. , 70 pages. https://doi.org/10.1109/IEEESTD.2008.4610935

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

[52] Intel. 2023. Math Kernel Liberary. https://software.intel.com/en-us/intel-mkl/.

[53] Olivier Chapelle Jean-Baptiste Tien, joycenv. 2014. Display Advertising Challenge. https://kaggle.com/competitions/criteo-display-ad-challenge Data: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[54] Dhiraj Kalamkar et al. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:1905.12322 [cs.LG]

[55] 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. In *IEEE TPDS*, Vol. 24. 1930–1940. https://doi.org/10.1109/TPDS.2012.290

[56] Mahmoud Abo Khamis, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, et al. 2020. Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies. In *TODS*, Vol. 45. Article 7. https://doi.org/10.1145/

3375661

[57] Hideaki Kimura, Vivek Narasayya, et al. 2011. Compression aware physical database design. In *PVLDB*. https://doi.org/10.14778/2021017.2021023

[58] Urs Köster et al. 2017. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *NeurIPS*. https://proceedings.neurips.cc/paper/2017/hash/a0160709701140704575d499c997b6ca-Abstract.html

[59] 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

[60] Alex Krizhevsky et al. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NeurIPS*. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[61] Arun Kumar et al. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD*. 16. https://doi.org/10.1145/2723372.2723713

[62] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. In *PACMMOD*, Vol. 1. Article 118. https://doi.org/10.1145/3589263

[63] 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

[64] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL] https://arxiv.org/abs/2002.11054

[65] Daniel Lemire. 2021. Number parsing at a gigabyte per second. In *Software: Practice and Experience*, Vol. 51. 1700–1727. https://doi.org/10.1002/spe.2984

[66] Daniel Lemire and Boytsov Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* https://doi.org/10.1002/spe.2203

[67] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, et al. 2019. Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent. In *SIGMOD*. 1517–1534. https://doi.org/10.1145/3299869.3300070

[68] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, et al. 2019. Tuple-Oriented Compression for Large-Scale Mini-Batch Stochastic Gradient Descent. In *SIGMOD*. 1517–1534. https://doi.org/10.1145/3299869.3300070

[69] 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

[70] Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. 2024. UniSparse: An Intermediate Language for General Sparse Format Customization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 99 (April 2024), 29 pages. https://doi.org/10.1145/3649816

[71] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, et al. 2018. Advances in Pre-Training Distributed Word Representations. In *LREC*.

[72] Anna Montoya, inversion, KirillOdintsov, and Martin Kotek. 2018. Home Credit Default Risk. https://kaggle.com/competitions/home-credit-default-risk

[73] Ingo Müller, Cornelius Ratsch, and Franz Färber. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*. https://api.semanticscholar.org/CorpusID:8114547

[74] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *IE2022*.

[75] NVIDIA. 2020. A100 Tensor Core GPU Architecture. images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[76] NVIDIA. 2023. CUDA Sparse Matrix lib. https://docs.nvidia.com/cuda/cusparse/.

[77] Dan Olteanu. 2020. The relational data borg is learning. In *PVLDB*, Vol. 13. 3502–3515. https://doi.org/10.14778/3415478.3415572

[78] 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

[79] Ismail Parsa. 1998. KDD Cup 1998 Data. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5401H.

[80] Arnab K. Paul, Jong Youl Choi, Ahmad Maroof Karimi, and Feiyi Wang. 2022. Machine Learning Assisted HPC Workload Trace Generation for Leadership Scale Storage Systems. In *HPDC*. https://doi.org/10.1145/3502181.3531457

[81] Karl Pearson. 1901. LIII. On lines and planes of closest fit to systems of points in space. In *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Vol. 2. 559–572. https://doi.org/10.1080/14786440109462720

[82] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011), 2825–2830.

[83] Tuomas Pelkonen et al. 2015. Gorilla: a fast, scalable, in-memory time series database. In *PVLDB*, Vol. 8. 1816–1827. https://doi.org/10.14778/2824032.2824078

[84] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. 2022. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. In *PVLDB*, Vol. 15. https://doi.org/10.14778/3551793.3551842

[85] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. 1426–1439. https://doi.org/10.1145/3448016.3452788

[86] Mercedes Piedra et al. 2019. Santander Customer Transaction Prediction. https://kaggle.com/competitions/santander-customer-transaction-prediction

[87] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. 2014. Database cracking: fancy scan, not poor man's sort!. In *DaMoN*. 4:1–4:8. https://doi.org/10.1145/2619228.2619232

[88] P. Ratanaworabhan et al. 2006. Fast lossless compression of scientific floating-point data. In *DCC*. 133–142. https://doi.org/10.1109/DCC.2006.35

[89] Alexander Ratner et al. 2017. Snorkel: rapid training data creation with weak supervision. In *PVLDB*, Vol. 11. https://doi.org/10.14778/3157794.3157797

[90] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data programming: Creating large training sets, quickly. In *NeurIPS*, Vol. 29. https://proceedings.neurips.cc/paper_files/paper/2016/file/6709e8d64a5f47269ed5cea9f625f7ab-Paper.pdf

[91] Walter Reade. 2019. Categorical Feature Encoding Challenge. https://kaggle.com/competitions/cat-in-the-dat

[92] Yousef Saad. 1990. SPARSKIT: a basic tool kit for sparse matrix computations - Version 2. https://api.semanticscholar.org/CorpusID:207974787

[93] Ricardo Salazar, Felix Neutatz, and Ziawasch Abedjan. 2021. Automated Feature Engineering for Algorithmic Fairness. *PVLDB* 14, 9 (2021), 1694–1702. https://doi.org/10.14778/3461535.3463474

[94] Sebastian Schelter et al. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

[95] Maximilian Schleich et al. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*. 16. https://doi.org/10.1145/2882903.2882939

[96] 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

[97] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, et al. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. In *PVLDB*, Vol. 10. https://doi.org/10.14778/3025111.3025125

[98] Shafaq Siddiqi et al. 2023. SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications. *PACMOD* 1, 3 (2023), 218:1–218:26. https://doi.org/10.1145/3617338

[99] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*. 17. https://doi.org/10.1145/3299869.3319854

[100] Mike Stonebraker et al. 2005. C-store: a column-oriented DBMS. In *PVLDB*. 553–564. https://dl.acm.org/doi/10.5555/1083592.1083658

[101] Michael Stonebraker, Paul Brown, et al. 2011. The architecture of SciDB. In *SSDBM*. 1–16. https://dl.acm.org/doi/abs/10.5555/2032397.2032399

[102] Julia Stoyanovich, Serge Abiteboul, Bill Howe, H. V. Jagadish, and Sebastian Schelter. 2022. Responsible data management. *Commun. ACM* 65, 6 (may 2022). https://doi.org/10.1145/3488717

[103] Felipe Petroski Such et al. 2020. Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data. In *ICML*, Vol. 119. 9206–9216. https://dl.acm.org/doi/10.5555/3524938.3525792

[104] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *KDD'08*.

[105] The pandas dev team. 2010. *Pandas 2.1*. https://doi.org/10.5281/zenodo.3509134

[106] Alessandro Ticchi et al. 2021. G-Research Crypto Forecasting. https://kaggle.com/competitions/g-research-crypto-forecasting

[107] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. 2019. Robustness May Be at Odds with Accuracy. In *ICLR*. https://openreview.net/forum?id=SyxAb30cY7

[108] John W. Tukey. 1962. The Future of Data Analysis. *The Annals of Mathematical Statistics* 33, 1 (1962), 1 – 67. https://doi.org/10.1214/aoms/1177704711

[109] Stef van Buuren and Karin Groothuis-Oudshoorn. 2011. mice: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software* 45, 3 (2011), 1–67. https://doi.org/10.18637/jss.v045.i03

[110] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, et al. 2014. DBDesigner: A customizable physical design tool for Vertica Analytic Database. In *ICDE*. 1084–1095. https://doi.org/10.1109/ICDE.2014.6816725

[111] Paroma Varma and Christopher Ré. 2018. Snuba: automating weak supervision to label training data. In *PVLDB*. https://doi.org/10.14778/3291264.3291268

[112] R. Vink and C. Peters. 2020. Polars: DataFrames for the new era. https://pola.rs/

[113] Maggie Demkin Walter Reade. 2019. Categorical Feature Encoding Challenge II. https://kaggle.com/competitions/cat-in-the-dat-ii

[114] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training deep neural networks with 8-bit floating point numbers. In *NIPS*. 7686–7695. https://dl.acm.org/doi/10.5555/3327757.3327866

[115] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The implementation and performance of compressed databases. In *SIGMOD*, Vol. 29. 55–67. https://doi.org/10.1145/362084.362137

[116] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. 2023. Register Tiling for Unstructured Sparsity in Neural Network Inference. *PLDI* 7, Article 188 (2023). https://doi.org/10.1145/3591302

[117] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. 2023. WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 920–934. https://doi.org/10.1145/3575693.3575742

[118] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. 2017. Online Deduplication for Databases. In *SIGMOD*. 1355–1368. https://doi.org/10.1145/3035918.3035938

[119] Huanrui Yang, Wei Wen, and Hai Li. 2020. DeepHoyer: Learning Sparser Neural Network with Differentiable Scale-Invariant Sparsity Measures. In *ICLR*. https://openreview.net/forum?id=rylBK34FDS

[120] Dongqing Zhang et al. 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *ECCV*. arXiv:1807.10029

[121] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. 2024. Compilation of Modular and General Sparse Workspaces. *Proc. ACM Program. Lang.* 8, PLDI, Article 196 (June 2024), 26 pages. https://doi.org/10.1145/3656426

[122] Hantian Zhang et al. 2017. ZipML: training linear models with end-to-end low precision, and a little bit of deep learning. In *ICML*. 4035–4043. https://dl.acm.org/doi/10.5555/3305890.3306098

[123] Jiujing Zhang et al. 2023. High-Ratio Compression for Machine-Generated Data. In *PACMMOD*, Vol. 1. Article 245. https://doi.org/10.1145/3626732

[124] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59–59. https://doi.org/10.1109/ICDE.2006.150