



Workload-aware Compressed Linear Algebra for Data-centric Machine Learning Pipelines

vorgelegt von
M.Sc.
Sebastian Baunsgaard

von der Fakultät IV - Elektrotechnik und Informatik
Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
-Dr.-Ing.-
genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr. Volker Markl
Gutachter: Prof. Dr. Matthias Boehm
Gutachter: Prof. Dr. Viktor Leis
Gutachter: Prof. Dr. Carsten Binnig

Berlin 2025

Abstract

Compression is an effective technique for fitting data in available memory, reducing I/O across the storage-memory-cache hierarchy, decreasing energy consumption, and increasing instruction parallelism. Modern machine learning (ML) systems exploit the approximate nature of ML and mostly use lossy compression via low-precision floating- or fixed-point quantized representations. The lossy techniques have an unknown impact on convergence and model accuracy compared to full precision training and create trust concerns. Furthermore, exploratory refinement of such lossy decisions is difficult to define in declarative ML pipelines. To use declarative language abstractions, lossless matrix compression applies lightweight compression schemes to numeric matrices and enables compressed linear algebra operations such as matrix-vector multiplications directly on compressed representations.

Traditional ML pipelines containing feature transformations and model training are increasingly extended into so-called data-centric ML pipelines with additional preprocessing steps for data cleaning, augmentation, and feature engineering to create higher-quality results. Given the trend towards increasingly complex composite pipelines, it is hard to infer the impact of compound ML pipeline primitives. Therefore, multiple variations of stacked techniques must be evaluated to find the best combinations. The evaluation of many pipeline variations is expensive but contains data redundancy that is exploitable via compression techniques. However, current compression techniques struggle to detect these redundancies.

Individual pipeline stages, such as data cleaning, augmentation, and feature transformations, collect data characteristics, such as distinct items, column sparsity, and column correlations. These properties are core components in selecting compression schemes. Current compression algorithms redundantly rediscover these statistics in compression planning while compressing pipeline intermediates. Some systems already exploit redundancy using sparsity exploitation on intermediates, which is a form of lossless compression. This thesis aims to evolve sparsity exploitation to general redundancy-exploiting lossless compression that exploits common values instead of only zero values. Existing work on lossless compression and compressed linear algebra enable such exploitation to a degree but face challenges for general applicability.

To solve the challenges, we introduce a workload-aware compression framework comprising a broad spectrum of new compression schemes to exploit different redundancy patterns and compressed kernels that can process long sequences of instructions with compressed intermediates and limited decompressions. The framework seamlessly fits into declarative ML pipelines by returning equivalent results to uncompressed linear algebra. We propose new feature transformation and engineering techniques that leverage information about the structural transformations collected in preprocessing pipelines. Furthermore, we develop a lightweight morphing technique adapting compressed intermediates to their subsequent linear algebra workloads. Instead of using a memory-centric approach that optimizes compression ratios, our workload-aware compression summarizes the workload of an ML pipeline and optimizes the compression scheme to minimize execution time.

All presented contributions are integrated into Apache SystemDS, an open-source ML system for the end-to-end data science lifecycle. We evaluate our implementation on micro benchmarks of components, end-to-end ML pipelines, and distributed federated linear algebra. Our evaluation shows asymptotic improvements in operations performed on workload-aware compressed data. The asymptotic changes and data size reduction translate to real-time gains on individual operations on real datasets up to 10,000x compared to uncompressed and 20,000x compared to previous compressed linear algebra. Furthermore, end-to-end ML algorithms improve by 6.6x, and a data-centric pipeline reduces power consumption by 3.6x.

Zusammenfassung

Kompression ist eine effektive Technik zur Unterbringung von Daten im Hauptspeicher, zur Reduktion von I/O entlang der gesamten Speicherhierarchie, zur Verringerung des Energieverbrauchs sowie zur Erhöhung der Instruktions-Parallelität. Moderne Systeme für Machine Learning (ML) machen sich die approximative Natur von ML zunutze und verwenden zumeist verlustbehaftete Kompression in Form quantisierter Gleit- oder Festkommazahlen mit verringerter Genauigkeit. Der Einsatz verlustbehafteter Techniken hat jedoch im Vergleich zu verlustlosem Training einen unbekannten Einfluss auf die Konvergenz und Genauigkeit von Modellen, woraus sich Vorbehalte bezüglich der Vertrauenswürdigkeit ergeben. Außerdem ist die explorative Verfeinerung solcher verlustbehafteten Entscheidungen im Kontext deklarativer ML-Pipelines schwierig zu definieren. Um deklarative Sprachabstraktionen nutzen zu können, wendet verlustlose Matrix-Kompression leichtgewichtige Kompressions-Techniken auf numerische Matrizen an und ermöglicht Operationen der linearen Algebra, wie Matrix-Vektor-Multiplikation, direkt auf den komprimierten Repräsentationen.

Traditionelle ML-Pipelines bestehend aus Feature-Transformationen und Modell-Training werden zunehmend zu so-genannten daten-zentrischen ML-Pipelines erweitert, welche zusätzliche Vorverarbeitungsschritte zur Daten-Bereinigung, Daten-Augmentation und zum Feature-Engineering enthalten, und somit die Qualität der Ergebnisse erhöhen. Angesichts des Trends zu immer komplexeren kompositen Pipelines ist es schwierig für Nutzende, den Einfluss zusammengesetzter ML-Pipeline-Primitive vorherzusehen. Daher ist eine Auswertung von Variationen zusammengesetzter Techniken erforderlich, um die beste Kombination zu finden. Die Auswertung einer hohen Anzahl von Pipeline-Variationen ist aufwendig, führt jedoch auch zu Redundanz in den Daten, welche durch Kompressions-Techniken und komprimierte lineare Algebra ausgenutzt werden kann. Existierende Kompressions-Techniken haben jedoch Schwierigkeiten diese Redundanzen zu erkennen.

Einzelne Pipeline-Stufen, wie Daten-Bereinigung, Daten-Augmentation und Feature-Transformation, ermitteln Dateneigenschaften, wie die unterschiedlichen Werte, Spalten-Sparsity und Spalten-Korrelationen. Diese Eigenschaften sind auch Kernbestandteile für die Auswahl von Kompressions-Techniken. Existierende Kompressions-Algorithmen berechnen diese Statistiken erneut im Zuge der Planung der Kompression von Zwischenergebnissen von Pipelines. Einige Systeme nutzen Redundanz bereits mithilfe von Sparsity-Exploitation-Techniken aus, welche eine Form von verlustloser Kompression darstellen. Das Ziel dieser Arbeit besteht in der Weiterentwicklung von Sparsity-Exploitation zu einer allgemeinen Ausnutzung von Redundanz basierend auf häufig vorkommenden Werten anstelle nur von Null-Werten. Existierende Arbeiten zu verlustloser Kompression und komprimierter linearer Algebra ermöglichen eine solche Ausnutzung bereits zu einem gewissen Grade, bringen jedoch Herausforderungen bezüglich der allgemeinen Anwendbarkeit mit sich.

Um diese Herausforderungen zu adressieren, führen wir ein workload-bewusstes Kompressions-Framework ein. Dieses umfasst ein breites Spektrum neuer Kompressions-Techniken zur Ausnutzung verschiedener Redundanz-Muster sowie komprimierte Kernel zur Verarbeitung langer Instruktions-Sequenzen mit komprimierten Zwischenergebnissen bei eingeschränkter Dekompression. Unser Framework fügt sich nahtlos in deklarative ML-Pipelines ein, indem es zu unkomprimierter linearer Algebra äquivalente Ergebnisse zurückgibt. Wir stellen neue Techniken für Feature-Transformation und -Engineering vor, welche sich in Vorverarbeitungs-Pipelines gesammelte Informationen über die strukturellen Transformationen zunutze machen. Außerdem entwickeln wir eine leichtgewichtige Morphing-Technik, welche komprimierte Zwischenergebnisse an ihre nachfolgenden Lineare-Algebra-Workloads anpasst. Anstelle eines auf die Optimierung der Kompressionsrate ausgerichteten speicher-zentrischen

Ansatzes fasst unser workload-bewusstes Kompressions-Framework den Workload einer ML-Pipeline zusammen und optimiert die Kompressions-Technik für die Minimierung der Ausführungszeit.

Alle präsentierten Beiträge sind integriert in Apache SystemDS, ein quelloffenes ML-System für den gesamten Data-Science-Lebenszyklus. Wir evaluieren unsere Implementierung mittels Mikrobenchmarks, durchgehenden ML-Pipelines sowie verteilten, föderierten Lineare-Algebra-Workloads. Unsere Evaluation zeigt asymptotische Verbesserungen für auf workload-bewusst komprimierten Daten ausgeführte Operationen. Diese asymptotischen Veränderungen und Reduktionen der Datengröße führen zu Realzeit-Gewinnen einzelner Operationen auf realen Datensets von bis zu 10,000x verglichen mit unkomprimierter und 20,000x verglichen mit existierender komprimierter linearer Algebra. Außerdem erzielen wir eine Verbesserung von durchgehenden ML-Algorithmen von 6.6x und reduzieren den Energie-Verbrauch einer datenzentrischen Pipeline um 3.6x.

Acknowledgements

I would like to thank all who helped me throughout my studies and in shaping this thesis. My academic journey has been like a marathon swim through challenging waters, and I want to reflect on it through the lens of a swimmer.

First, I would like to thank Pınar Tözün, who motivated me to pursue a PhD and, like a talent scout at the poolside, recognized potential in me before directing me to my PhD advisor, Matthias Boehm. I am grateful to Matthias Boehm, my swimming instructor with high expectations, for teaching me to navigate academic waters while giving me freedom to develop my own style. I am thankful for Matthias entrusting the topic of compressed linear algebra to me—a challenging but rewarding swimming lane. I would especially like to thank his patience when he took the time to listen to my sometimes crazy ideas, which he, with his experience, shaped into a coherent direction, much like a coach refining a swimmer's technique for optimal performance.

Second, I would like to thank my colleagues, my fellow student swimmers, who kept my time as a PhD student fun and entertaining, even during the most gruelling training sessions in our shared academic pool. I would like to start with thanking Sebastian Benjamin Wrede, for making the daring move from Denmark to Austria with me—a brave dive into new waters. Starting my PhD in TU Graz was made enjoyable by swimming alongside Arnab Phani, Mark Dokter, Shafaq Siddiqi, Saeed Fathollahzadeh, and Patrick Damme. A special thanks goes to Arnab, who has listened to all my complaints over the years, like a fellow swimmer always ready with a buoy, and to Patrick for counselling on technical details—helping me in the deepest parts of my swimming lane. I also thank my colleagues at TU Berlin: Christina Dionysio, Philipp Ortner, Carlos Muniz Cuza, David Justen, Ramon Schöndorf, and Grigori Turchenko, for their fresh perspectives from different swimming lanes.

Third, I would like to thank my students, who challenge my conceptions on understanding and bring me joy when surpassing me and my ideas. They are the new swimmers I've had the privilege to guide, who often surprise me by mastering strokes I'm still perfecting myself.

Fourth, I would like to thank my parents, especially for supporting my sometimes risky moves from country to country. They have been my constant supporters on the poolside, watching as I ventured into deeper waters. I still fondly remember driving to Graz with all my things, without having an apartment yet—essentially jumping into the deep end without knowing if there would be a lifeguard.

Last but not least, to Olga Ovcharenko, my partner in crime and best training companion—thank you for motivating me through rough waters, reading my unreadable drafts, and setting the pace when deadlines were out of sight.

As I emerge from this academic pool, I carry not just a degree but the endurance and confidence of a seasoned swimmer, thanks to excellent coaching and supportive teammates.

Table of Contents

Abstract	iii
Zusammenfassung	v
Acknowledgements	vii
1 Introduction	1
1.1 Motivation and Challenges of Compressed Linear Algebra	2
1.2 Potential Analysis	3
1.2.1 Distinct Values	4
1.2.2 Lossy Transformations	4
1.2.3 Non-numerical Data	5
1.2.4 Correlation	5
1.2.5 Pre-processing Time	6
1.2.6 Energy Efficiency	6
1.3 Related work of Workload-aware CLA	7
1.4 Contributions and Structure of the Thesis	9
2 Background	11
2.1 Lossless Compression Techniques	12
2.2 Categorization of Compression Techniques	18
2.3 Compression Systems / Compound Techniques	18
2.4 Compressed Linear Algebra	19
2.4.1 CLA in SystemML	20
2.4.2 Tuple-oriented Compression	24
2.4.3 Grammar-compressed Linear Algebra	26
2.4.4 Huffman Address Map	27
2.4.5 Compressed Shared Elements Row	27
2.5 Workload Adaptation	28
2.5.1 Cost Modelling	28
2.5.2 Property Estimation	29
2.5.3 Multi Objective Optimization	30
2.6 Feature Engineering	30
2.6.1 Categorical Transformations	31
2.6.2 Arithmetic Transformations	31
2.6.3 Feature Augmentations	32
2.7 Summary	34
3 Workload-aware CLA	35
3.1 Compression	37
3.1.1 Dictionary-based Compression	37
3.1.2 Index Structures for Compression	38
3.1.3 Online Compression Sequence	38

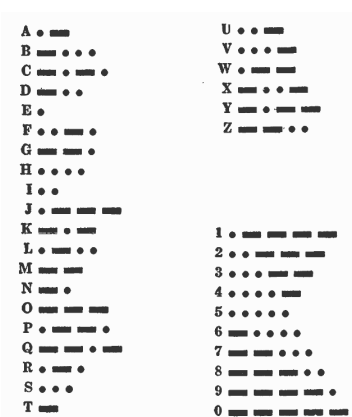
TABLE OF CONTENTS

3.1.4	Compressed Design Principles	41
3.2	Compressed Operations	42
3.2.1	A Motivating Operations Example	42
3.2.2	Overlapping Column Groups	43
3.2.3	Aggregations on Compressed Data	44
3.2.4	Binary Element-wise Operations	45
3.2.5	Slicing	46
3.2.6	Matrix Multiplication	46
3.2.7	Decompression	52
3.3	Workload-based Adaptation of Compression Plans	52
3.3.1	Workload Trees	52
3.3.2	Compiler Integration	54
3.3.3	Workload-dependent Runtime Compression	54
3.4	Experiments	55
3.4.1	Experimental Setting	55
3.4.2	Compression Performance	55
3.4.3	Operations	56
3.4.4	Operation Sequences	59
3.4.5	Workload Cost Analysis	60
3.4.6	End-to-end Algorithms	60
3.4.7	Hyper-parameter Tuning	62
3.4.8	Comparison with Other Systems	63
3.5	Summary	64
4	Compressed Transformations	65
4.1	Heterogeneous Frame Compression	66
4.1.1	Compressed Frame Design	66
4.1.2	Compressed Feature Transformations	67
4.1.3	Compressed Feature Engineering	70
4.2	Morphing	70
4.3	Compressed I/O	71
4.3.1	Compressed Data Layout	72
4.3.2	Reading and Writing Compressed Data	72
4.4	Compiler and Runtime Integration	73
4.5	Experiments	74
4.5.1	Experimental Setting	74
4.5.2	Frame Compression and I/O	75
4.5.3	Compressed Feature Transformations	76
4.5.4	Compressed Word Embeddings	77
4.5.5	ML Algorithm Performance	78
4.5.6	Data-centric ML Pipeline	80
4.5.7	Comparisons with Other Systems	80
4.6	Summary	80
5	Asynchronous Compression in Federated Learning	81
5.1	Federated Primitives	82
5.1.1	Federated Data	82
5.1.2	Federated Privacy	83
5.2	Federated Runtime	83
5.2.1	Federated Backend	83
5.2.2	Federated Linear Algebra	85
5.2.3	Federated Data Preparation	87
5.2.4	ExDRA Future Work	87

5.3	Standing Federated Workers	88
5.4	Experiments	89
5.4.1	Experimental Setting	89
5.4.2	ML Algorithms Performance	90
5.4.3	ML Pipelines Performance	92
5.4.4	Asynchronous Compression	92
5.5	Summary	96
6	Conclusions and Future Work	97
6.1	Summary	97
6.2	Conclusions	98
6.3	Future Work	98
	Bibliography	118
	Appendix A	119
A.1	Arithmetic Encoding	119
A.2	Sparse Operations Analysis	121
A.3	CSR-VI Operations Analysis	128
A.4	Kernel Function Example	129

1

Introduction



A	· —	U	· · —
B	· · · —	V	· · · —
C	— · ·	W	· — —
D	— · ·	X	— · · —
E	·	Y	— · — ·
F	· · · —	Z	— — · ·
G	— · —		
H	· · · ·		
I	· ·		
J	· — — —		
K	— · —	1	— · — — —
L	· — · ·	2	· · — — —
M	— —	3	· · · — —
N	— ·	4	· · · · —
O	— — —	5	· · · · ·
P	— · — ·	6	· · · · ·
Q	— — · —	7	— · · · ·
R	· — · ·	8	— — · · ·
S	· · ·	9	— — — · ·
T	—	0	— — — — —

Figure 1.1: Morse Code [214]

An early example of data compression is Morse code, invented in 1838 [245]. Morse code’s compression scheme is constructed of multiple layers. The binary signal of on and off contains a sequence of ternary base building blocks called dots, dashes, and pauses of varying lengths. Letters can be deciphered from the ternary signal via codewords. To shorten the duration of sending messages and, in turn, the compression ratio, the codewords were designed such that the duration of sending frequent letters is low, e.g., ‘e’ uses a single dot.

Modern Compression: In modern data management and machine learning (ML) systems, compression is a well-established and effective technique for fitting data in available memory, reducing I/O, energy [109], and memory bandwidth usage [194, 74], and increasing instruction parallelism [243]. Data management systems with declarative interfaces almost exclusively rely on lossless compression to ensure correct results and leverage lightweight techniques [59, 56] that offer a balance of compression ratios and (de)compression speed that determine operational performance.

ML and Compression: In contrast to lossless compression in data systems, ML systems exploit the approximate nature of ML—especially for mini-batch training of deep neural networks (DNN)—and mostly use lossy compression such as quantization (i.e., static or dynamic discretization) [91, 259], sparsification (clipping of low quantities) [93, 176], new data types (e.g., `bfloat16`, `TF32`) [202, 128, 176], dimensionality reduction [116] and sampling (few step/epoch mini-batch training [221], or sampled batch training [181]). An extreme example is 1-bit quantization of gradients [209]. However, lossy compression introduces unknown behavior on new datasets and models, which creates trust concerns and requires an exploratory trial-and-error process to find the right settings and sometimes tradeoff time for accuracy [238]. Therefore, any lossy decisions in ML have to be set by users or automatically evaluated.

ML Pipelines: Modern machine learning (ML) training comprises more than choosing ML algorithms or neural network architectures, and hyper-parameters. Data-centric ML pipelines extend traditional ML pipelines of feature transformations and model training by additional preprocessing steps for data validation [206, 95], data cleaning [212], feature engineering [203], and data augmentation [196, 197, 232, 131] to construct high-quality datasets with good coverage of the target domain. These techniques can substantially improve model accuracy [131, 212, 248, 67], generalizability [7], and other measures like fairness [203, 219].

1.1 Motivation and Challenges of Compressed Linear Algebra

There are several motivational factors for using compressed formats in linear algebra operations. However, remaining challenges from related work hinder broader adoption. This section outlines these characteristics in the context of linear algebra-based data-centric ML pipelines.

Sources of Redundancy: The iterative nature of finding good data-centric ML pipelines causes both operational redundancy (e.g., fully or partially repeated preprocessing steps) [190] as well as data redundancy [23]. The data redundancy is present in similarly compressible patterns across pipeline allocations from small cardinalities, repeated values, sparsity, and column correlation. Besides this natural data redundancy, data-centric ML pipelines create additional redundancy. Examples are new data points or feature modifications and systematic transformations such as missing value imputation by mean or mode and data cleaning by robust functional dependencies [66]. While beneficial for model quality, iterative selection of such data-centric ML pipelines is expensive. Exploiting the data redundancy in pipelines is appealing if compressed allocations can minimize redundancy though pipeline components and support pipeline workloads without decompression.

Lossless Redundancy-exploitation: A common approach for exploiting data redundancy without quality degradation is lossless compression. Sparsity exploitation is a classic redundancy-exploitation technique and is currently a major trend across the stack from hardware [176, 178], over systems [34, 156, 215], to algorithms [93, 81, 199]. It exploits sparsity (many zero values) in data representations, typically in the form of matrices or tensors by avoiding processing zero values via dedicated data layouts, sparse operators, and even ML algorithm formulations [251]. We can observe sparsity in many real-world use cases, such as traffic flow, web page ranking and user ratings of products. Sparsity also naturally occurs in many linear algebra programs, for instance, when solving partial differential equations or performing feature engineering via one-hot encoding [215]. However, sparsity-exploitation is limited to zeros and does not exploit other sources of redundancy. While sparsity is a dominating topic in the field, compression techniques that apply lightweight database compression schemes can exploit other redundancies, such as a low number of distinct values or long sequences of repeated values, while also supporting linear algebra operations directly on the compressed representation, also called Compressed Linear Algebra (CLA) [74, 147, 77, 241, 130].

The Problem: Despite Compressed Linear Algebra’s compelling property of executing operations directly on compressed representations, some limitations and missing functionality hinder general applicability and, therefore, are problems addressed in this thesis. To help identifying these problems, we start with the motivation and goals from CLA [74, 75, 73].

The Original Goal:

There was two optimization objectives of the original implementation of CLA [74]. ① Compressed linear algebra operations performance. ② Good compression ratios to fit larger datasets into memory. Figure 1.2 shows the consequence of improving

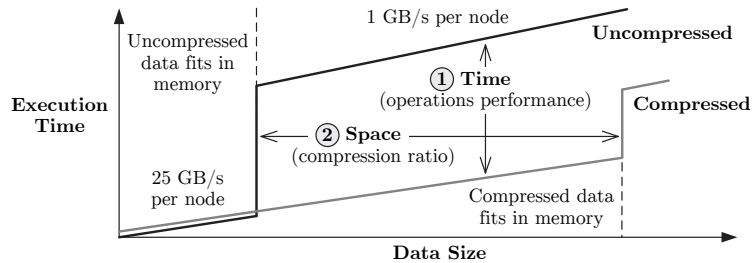


Figure 1.2: Goals of CLA [73]

these objectives. The x-axis signifies data size growth, and the y-axis is theoretical execution time. The figure shows that when data fits in memory, the memory bandwidth of processing data is high at 25 GB/s. However, the bandwidth drops to 1 GB/s once the data spills to disk. While the numbers are not the same in newer systems, the theory applies both to local execution with vertical jumps in execution time on total local memory and distributed bounded by total aggregate cluster memory. Optimizing for ② enables CLA to keep more data

in memory, delaying the need to spill intermediates to disk. Optimizing for ① improves the operational gains relative to the uncompressed performance once the uncompressed data no longer fits in memory. Importantly, the figure also shows that once the compressed data is spilling to disk, compressed execution expects better scaling than uncompressed execution because compressed data is less bottlenecked by I/O.

Local Focus: While these goals are important, they miss exploiting the single-node local potential of CLA, which can be better than uncompressed operations. However, we need to solve a few challenges to acquire this improvement.

Compressed Operations & Structures: Perhaps obvious, the first key challenge is designing compressed structures that allow efficient execution of operations, preferably without decompression. The compressed operations should be faster than their uncompressed matching operations. CLA’s performance target of individual operations was similar to uncompressed linear algebra when the uncompressed data fit in memory. However, it is key for CLA adaptation to have faster operations in compressed space for in-memory execution. If the operations are not quicker, then solving the rest of the challenges would have less impact, and CLA would mainly be applicable when data does not fit in memory. Another challenge for operational performance is finding compression schemes excellent at exploiting dense compressed matrices or sparse compressed matrices that encounter densifying operations. Many current CLA techniques [241, 157, 77, 147, 73, 74] do not support this use case of which most default to decompression.

Compressed Intermediates and Sequences: Another challenge is to extend the number of operations performed on compressed intermediates. Operations should aim towards returning compressed results to solve this challenge. While there previously was some support for long sequences of operations in compressed space, extending this capability as far as possible should improve the overall performance. For instance, if the compressed data can be read from disk without decompression it would be possible to skip the compression and start processing directly. Furthermore, if possible, operations should return compressed representations with minimal changes and maximum reuse of input data structures of compressed data.

Workload-aware Compression: A final challenge, and perhaps most importantly, the framework should automatically choose where to introduce compression in programs and what compression techniques to use based on the workload. Similar to specializing data structures for specific algorithms and workloads, we should automate the data layout of compressed data, choosing encoding schemes that optimize for the performance of a given workload. A holistic workload adaptation would include the online compression costs, the linear algebra-based computational workload, and data access patterns of operations to improve the I/O throughout the memory hierarchy.

Compressibility: While solving these challenges does improve the potential of using CLA in more use cases, it does not matter if there is no potential to exploit redundancy in data. Therefore, to ground the research, we performed a potential analysis.

1.2 Potential Analysis

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.

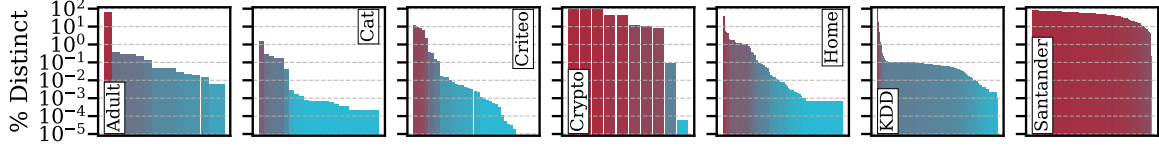


Figure 1.3: Relative Number of Distinct Values in Datasets.
Columns Sorted by the Number of Distinct Values.

1.2.1 Distinct Values

The number of distinct values d is a common statistic to exploit in compression [57, 168], especially in string columns. Dense Dictionary Compression (DDC) [74, 23, 168, 240, 18], also called DICT [61] in the database community, constructs a dictionary of d values and encodes values as integers referencing positions in the dictionary. Figure 1.3 shows several datasets with their ratio of d to rows per column. Some columns contain less than 0.001% distinct values. Compressed operations that exploit the distinct values can reduce execution time in such cases by 99.999% [23]. Unfortunately, data does not always contain a low number of unique values, motivating additional compression techniques.

1.2.2 Lossy Transformations

Feature engineering can reduce the number of distinct values d via lossy transformations such as binning, feature hashing, or quantization. In binning, we distinguish static and learned [258, 257] quantization schemes. An example of a static scheme is equi-width quantization, 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. In contrast, learned schemes [257] use various techniques—such as quantiles or neural networks—to find optimal quantization boundaries (smaller bins for high-frequency value ranges). Equi-height quantization maps input values to buckets by Δ quantiles. Figure 1.4 shows the relative loss of equi-width and equi-height quantization. The x-axis varies Δ and the y-axis shows the mean absolute error: $MAE(\mathbf{X}, \hat{\mathbf{X}}) = \sum_1^{|\mathbf{X}|} |x_i - \hat{x}_i| / |\mathbf{X}|$. The upper and lower bounds of the blue and orange colored areas are the min/max absolute errors. Quantization and incurred errors show a linear relationship (log-scale plots) of roughly $MAE(\mathbf{X}, Q_{\Delta}(\mathbf{X})) \approx 2 \cdot MAE(\mathbf{X}, Q_{2\Delta}(\mathbf{X}))$, meaning if Δ doubles, the MAE error is halved. Learned schemes can improve the MAE using higher Δ or optimize for other goals such as model accuracy [257], compression size, or Pareto-optimal combinations.

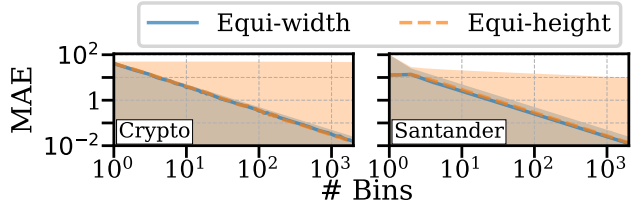


Figure 1.4: Lossy Quantization Effect on Values.

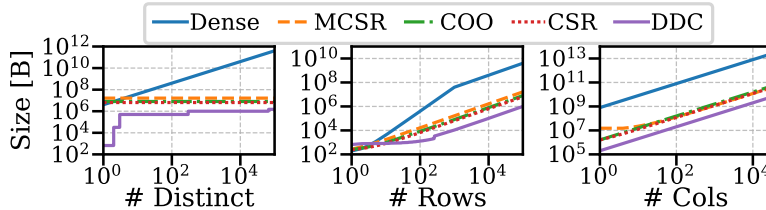


Figure 1.5: Output Memory Sizes of One-Hot/Dummy Coding an Input.

1.2.3 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 [63] and stemming. Figure 1.5 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 1,000 distinct values, 100k rows and 5 columns). The output shape is $[\text{\#rows}, \text{\#columns} \cdot d]$, and sparsity is $1/d$. The y-axis shows the in-memory size in bytes of the encoded output matrix. Using a dense matrix for one-hot-encoding shows worse performance in all cases beyond very few distinct values or rows. Sparse layouts such as Compressed Sparse Rows (CSR) [22], COOrdinate matrix (COO) [22], and Modified CSR (MCSR) [33] yield good compression in all cases. Sparsity exploitation performs exceptionally well when scaling d . However, compared to a DDC compression, all the other solutions allocate more memory. The dampened size increase for Dense in the middle plot is because the d increases until 1000 and not above.

1.2.4 Correlation

Another property that impacts compressibility is the correlation between columns. Figure 1.6 shows the relative increase in the number of distinct tuples when combining 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 be the number of distinct values in column i and $d_{i,j}$ the number of distinct tuples of co-coded columns i and j . Then, each cell $c_{i,j}$ in the figure shows $c_{i,j} = 2d_{i,j}/(d_i + d_j)$ to demonstrate the relative increase of values contained in tuples by combining the columns. White (with $c_{i,j} = 1$) indicates columns with perfect correlations. Perfect correlation can only happen when two column groups have the same d as in $d_i = d_j$, and when combined, $d_{i,j}$ does not increase. Perfect correlation happens, for instance, in the case of all pairs of one-hot-encoded columns originating from the same column. Any other combination of columns where $d_i > d_j$ has a minimum bound of $d_{i,j} = \min(d_i, d_j)$. In the case where $d_i > d_j$ and all values of d_i only correlate with a single distinct value (a many-to-one relationship) in d_j , $c_{i,j}$'s minimum bound is $2d_i/(d_i + d_j)$. The upper bound of any combination is $c_{i,j}$ is $2d_i d_j / (d_i + d_j)$, indicating all pairs of distinct values occur when combining the columns. Co-coding algorithms use $c_{i,j}$ and other properties to determine if combining columns into single encodings is better than leaving them apart in separate compression schemes.

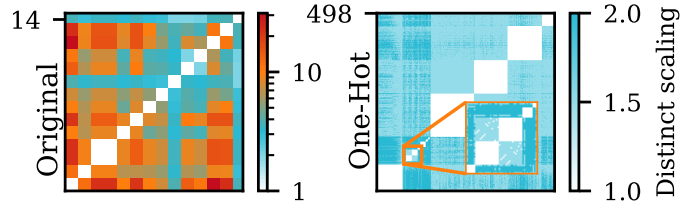


Figure 1.6: Relative d Increase Co-coding Features in Adult: Original and One-Hot-Encoded Features.

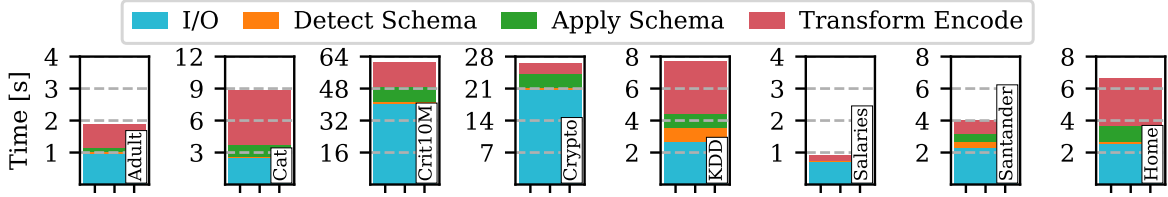


Figure 1.7: Cost Breakdown of Reading, Detecting Schemas and Applying Schemas, and Losslessly Encoding Different Datasets.

The Co-coding Problem: A greedy co-coding algorithm requires $\mathcal{O}(m^2)$ time (where m is the number of columns) to discover these correlated columns. Ideally, co-coding would first group one-hot-encoded features with perfect correlation and subsequently other correlated features. Rediscovering the correlation between columns is non-trivial and potentially very expensive since each combination of columns has to be analyzed (potentially on a sample). 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 favour of fast compression. Interestingly, Figure 1.6 shows a perfect correlation between the original features 3 and 4, while the one-hot-encoded version does not perfectly co-code on all sub-combinations of those columns (see zoomed-in area). Instead, this perfect correlation is only detectable via evaluations of larger sub-groups. Therefore, pushing compression through feature transformations has the potential for both runtime reduction and improved compression.

1.2.5 Pre-processing Time

Figure 1.7 shows the execution time of pre-processing the different datasets. We read datasets in CSV format 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 matrices through feature transformations. There is potential to improve all these stages via compression. First, reading compressed representations from disk reduces reading overheads. Second, frames saved with a schema can skip value type detection operations. Third, compressed formats that directly support feature transformations could improve performance. Therefore, a combined pre-processing and compression analysis could significantly reduce the end-to-end execution time depending on the algorithm(s) and data used. We further extend this train of thought with new ideas on feature transformation techniques that can process compressed inputs directly without decompression to reduce execution time and memory consumption.

1.2.6 Energy Efficiency

Transporting data from remote machines, from storage, or even from RAM into cached memory consumes a lot of energy [109]. The actual operations only draw a tiny fraction of the overall power. Therefore, if *high energy efficiency is required, the application must have very good data locality* [109]. As an

example, Figure 1.8 shows the overhead of an instruction fetching data from a CPU cache L3 to register processing. In the example, the actual operation takes 0.3 nJ and the transfer dominates with 7.9 nJ. If the instruction loads data from RAM, the cost increases by more orders of magnitude [109]. By compressing data, we can reduce the amount of data transported through the memory hierarchy, improving data locality and reducing the overall energy consumption.

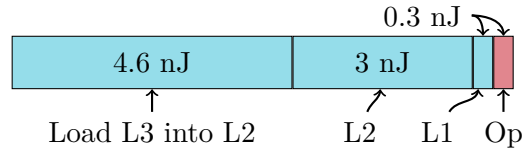


Figure 1.8: Energy Costs of Addition Operation (64 bit) in CPU Cache [160].

1.3 Related work of Workload-aware CLA

Workload-aware CLA connects to multiple related fields, including Lossy and lossless compression of matrices and heterogeneous frames, database compression and physical design, and techniques for data reorganization and adaptation of allocations with semantic equivalence.

Lossless Matrix Compression: Naturally, the closest area of related work is lossless matrix compression. General-purpose data-parallel frameworks like Spark [255] or Flink [16], scientific data formats like NetCDF and HDF5 [97], and storage managers like SciDB [217] and TileDB [180] also support compression, but decompress block- or partition-wise for operations. Sparsity exploiting compression had full software package support already in 1990 with SparseKit [201] using specialized data structures to exploit non-zero values. Sparsity exploitation is now commonly supported in most linear algebra frameworks such as IntelMKL (now behind OneAPI) [117] and CuSparse [175] but still in active research [215, 125, 242, 127]. Early work includes traditional sparse matrix representations (e.g., CSR, CSC, COO) [201] while more recent work in e.g., TACO [127] automatically compiles sparse kernels for tensor linear algebra. Sparsity exploiting compression techniques by Kourtis et al. already leveraged dictionary coding [130, 124], as well as delta and run-length encoding [124]. Subsequent work on compressed linear algebra (CLA) focused on online compression and entire ML algorithms, where CLA [74, 75] uses column compression for batch algorithms, and TOC [147] uses tuple compression for mini-batch algorithms. Other works exploit different properties such as integer time series values [30], floating point time series [150], and bounded ranges of floats [152]. Recent work on grammar-compressed matrices reports operation performance proportional to the compressed size, specifically the CSRv representation [77], while others presented impossibility results (worst-case) for efficient matrix-vector multiplications on grammar-compressed matrices such as Lempel-Ziv [6]. Factorized learning [133, 207] further pushes operations of ML training algorithms through joins and can be seen as a specialized form of lossless compression exploiting available schema information [177] to avoid materializing denormalized tables. These factorization ideas can also be implemented on top of ML systems [50] by representing joins via structured selection matrices. The most studied type of compression is integer-based compression [134, 57, 60, 142] while floating points with exponent and mantissa pose some difficulties. XOR [195] used by Facebook in Gorilla [187] is a technique specific for floating point compression, followed by advancements in Chimp [150] and even more recently in ALP [10].

Lossy Compression and Sampling: In the context of mini-batch DNN training and scoring, there is a broad adoption of lossy compression. First, quantization discretizes floating-point into fixed-point representations such as UINT8 for scoring [91]. Mainstream ML systems rely mostly on homogeneous lossy compression because they retain regular, dense data access. First, a common choice is uniformly encoding and processing all floating point values in reduced precision [115]. Many systems explore uniform encodings via techniques like mantissa truncation [5, 27] and new data types with different trade-offs of range and precision [176]. Examples include Google’s bfloat16 (1+8+7 bits) [202, 44, 123], Intel’s Flexpoint (shared subset of exponent bits) [128], and NVIDIA’s TF32 (1+8+10) [176]. Another common technique is static min/max binning (equi-width) [91] and learned quantization schemes (equi-height via quantiles) [259, 261]. Such quantization schemes are also used for efficient data transfer in ZipML [259] and SketchML [120]. With residual accumulation at the workers, some systems reduce communicated values to a single bit [209, 110]. The challenges of training with low 8-bit FP precision are addressed with chunk-based accumulation and stochastic rounding [237]. Other leverage sparsification or value clipping (omit small values) [93, 176], dimensionality reduction like auto encoders [116] and classic algorithms such as PCA [184] and t-SNE [107], sampling in BlinkML [181], DNN activation compression in COMET [121], sampling or coresets [11] to train with fewer mini batches [221], and progressive compression schemes [238, 132]. Unfortunately, the unknown impact on results creates trust concerns, requires trial and error, and is problematic for declarative ML pipelines. Recent work in MLWeaving [238] introduced data structures for efficiently extracting different granularities

for simplifying exploration, while BlinkML [181] estimates the minimum sample size to satisfy an accuracy constraint.

Database Frame Compression: There is a large body of work in compressing heterogeneous data in the data management literature. Tabular data in databases is commonly compressed in schemes that exploit entire columns having the same datatype [1, 218]. Systems most commonly employ variations of five basic encodings techniques [57, 60]: Frame-of-reference (FOR) [88], delta encoding (DELTA) [264], dictionary encoding (DDC) [168, 240, 18] in database compression more commonly called DICT [61], run-length encoding (RLE) [3] and null suppression (NS) [142, 240]. BtrBlocks [134] is a recent work showcasing the effectiveness of nesting these simple compression techniques with highly efficient SIMD decompression. White box compression [85] similarly apply recursive compression. General-purpose heavy-weight compressors are also applied to compress any modality of data. Examples include Snappy [90] and Zstd [76]. Most systems support storing their data in various compressed formats that can combine multiple encoding schemes. Examples are Parquet [52], HDF5 [97], and SciDB [216] for storage, as well as Arrow [53] for transfer, e.g., Arrows DDC encoded data [54]. There are many dedicated compression techniques for specific value types, for instance, for strings in Pattern-based Compression (PBC) [260] that decomposes string values via entropy encoding (Huffman coding [111, 194]), and FSST [41]. These fine-grained methods share some commonalities but are orthogonal techniques that, in many cases, can be stacked like in BtrBlocks [134].

Workload-aware Physical Database Design: Work on lossless matrix compression like CLA [74, 75] and TOC [147] was inspired by lossless compression in column stores and related query processing on compressed data. Extensions include patched encoding schemes (separate handling of exception values) [264], order-preserving dictionary coding [28, 153], and exploitation of such schemes in query processing [193, 28, 136, 21]. The handling of default values used in this thesis is also related to header compression in SAP HANA [204], and the fast-mode column add in Teradata [223]. The performance/compressed-size tradeoffs of existing schemes are, however, strongly data-dependent [108, 59]. For that reason, existing systems largely rely on conservative selection heuristics [136, 3, 2], but there is also work on cost modelling [59, 38, 48], and balancing query performance and storage size with different column group projections and encodings [230]. Some adapt similarly based on sparsity [14]. Many systems rely on combining cost modelling of compressed size and query performance [231, 39, 48, 62, 126] and many techniques rely on an offline compression for selection and adapting to workloads [37]. Some systems generates workload traces to use for compression decisions [183]. Once compression choices are reflected in the costs, they influence what-if physical design tuning. Compression-aware design tuning [126] and FITing-Tree [82] showed how index compression can affect index selection choices, and learned partitioning schemes maximize partitioning pruning [253] (e.g., via small materialized aggregates [162]). Furthermore, recent work introduced memory-budget-constrained offline compression for selecting encoding schemes based on estimated costs and compression ratios [36], and related data partitioning across storage tiers [139, 234].

Data Reorganization: Data reorganizing, also called morphing, is closely related to this thesis. Prior work like database cracking by Idreos et al. [192, 102, 114, 113, 112] also dynamically reorganizes data based on query workload. Other work dynamically chooses (1) the physical design of storing data [20], (2) where to place tuples on distributed servers (e.g. Clay [210]), and (3) online deduplication of stored blocks [250]. MorphStore [57, 101] proposes a morphing wrapper enabling on-the-fly recompression of intermediate results with lightweight compression schemes for relational algebra.

1.4 Contributions and Structure of the Thesis

This thesis addresses the challenges of compressed linear algebra by extending and combining related work and developing new compressed operations with the following contributions.

Background: [Chapter 2](#) clarifies the background and introduces prerequisite knowledge to understand the contributions to compressed linear algebra. The chapter separates others' work from this thesis by including an extensive section that covers the previous work's techniques and approaches to performing compressed operations. The covered topics include a detailed walkthrough of lightweight, heavyweight, lossless, and lossy compression techniques focusing on techniques applicable to compressed operations. We continue with existing algorithms and frameworks for performing operations directly on compressed formats. Followed by data-centric ML pipeline components focusing on feature engineering of numeric matrices and transformations of heterogeneous datasets. Finally, we cover the basics of federated learning architectures for distributed computation on federated data.

Workload-aware Compression: In [Chapter 3](#), we introduce the design and implementation of workload-aware CLA aimed at compressed operation performance better than uncompressed while achieving good compression ratios to fit larger datasets in memory and reduced I/O. The internal objective for selecting compression schemes focuses on minimizing execution time while factoring in compression, compressed operations, conditional control flow of programs, and potential I/O to better adapt to data, operation, and cluster characteristics. We describe new compressed operation strategies and techniques focusing on delaying computation and reusing compression structures in operations to exploit *redundancy*. These compressed operations can, in many cases, return compressed intermediates, improving the operational performance relative to the number of distinct values.

A Case for Compressed Preprocessing: Feature transformations encode heterogeneous categorical and numerical features into purely numerical matrices [189]. This conversion is a rich source of information about 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. Data-centric ML pipelines iteratively evaluate several permutations of engineered features and different transformations. Therefore, we make a case for *pushing compression through feature transformations and feature engineering to the sources* in [Chapter 4](#). Holistic support requires compressing heterogeneous frames in a form amenable to compressed feature transformations and compressed I/O without decompression. Since data and workload characteristics of enumerated ML pipelines may differ, there is a need for morphing [101, 62] compressed intermediates into workload-optimized representations [23].

Asynchronous Federated Compression: While our focus is compressed linear algebra, we also introduce contributions to federated ML training and scoring in [Chapter 5](#). Federated ML typically consist of a parameterserver setup with a centralized aggregation of model weights on a *server* and distributed training on *federated workers*. A key objective of federated ML is to train centralized models without consolidation of federated data. The chapter describes a federated backend extension to SystemDS that supports parameterserver workloads and linear algebra programs. Interestingly, federated workers' workloads fit nicely with asynchronous compression because it allows workers to adapt their data allocations to their specific workloads in low utilization periods between federated requests.

2

Background

Algebra studies the generalization of arithmetic operations and has been leveraging various representations since time immemorial. These representations together with algorithms allow us to perform calculus to help understand the world around us.

History: An interesting early example is Rhind Mathematica papyrus [12] by Ahmose, who coincidentally is the first math author we know of. The papyrus contains a handwritten dialect of hieroglyphs to show recipes for various algebraic tasks similar to current definitions of programs to compute our linear algebra operations. It even contains an approximation of $\pi = 4(8/9)^2 = 3.16$. However, the efficiency of calculation using the hieroglyphs, in many ways equal to Roman calculus, is slower than the current decimal numeral system because large numbers were difficult to calculate with [154].

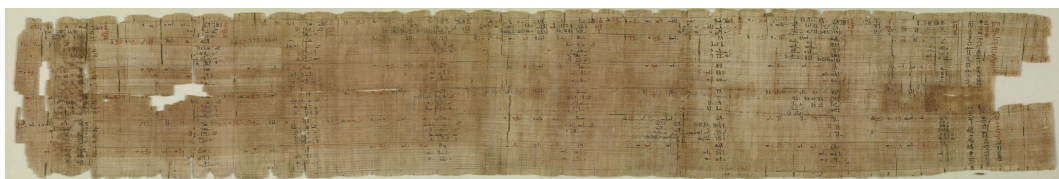


Figure 2.1: Rhind Mathematica Papyrus [12], CC BY-NC-SA 4.0 [13]

Modern: Modern ML systems use abstractions to perform linear algebra operations on scalar, vector, matrix, or higher-dimension tensors while leaving the underlying data structures up to systems' implementations [31]. This decoupling from underlying data allows systems to optimize the data representations together with algorithms. Improving the adaptability of algorithms and data for linear algebra instruction sequences.

System Integration: As an example, SystemML [86, 32, 35, 227] compile entire programs via a high-level declarative language abstraction of linear algebra instructions with R-like syntax. The design allows users to declare their intent without defining specific underlying runtime allocations or execution modes. Underneath the declarative language abstraction, the system automatically decides on runtime strategies such as parallelizing loop bodies [35], using distributed operations [35], and/or leveraging hardware accelerators [188]. The system chooses specialized data structures such as arrays, trees, sparsity exploiting matrices [201], or specialized formats such as compression [74, 147, 41, 143, 135]. Data structures can be further specialized with different value types, e.g., floats, strings, chars, or integers.

Outline: This chapter tries to cover the background of the contributions of the thesis. We start with a brief cover of influential lightweight compression schemes and techniques in Section 2.1 explaining elements of compressing dense matrices through sparse compression and standard to complex related lossless compression techniques. After the lossless background, we cover various existing compressed linear algebra frameworks in Section 2.4. Existing work on adapting compression to workload in Section 2.5, before finalizing the chapter with a summary of standard ML preprocessing and feature engineering in Section 2.6.

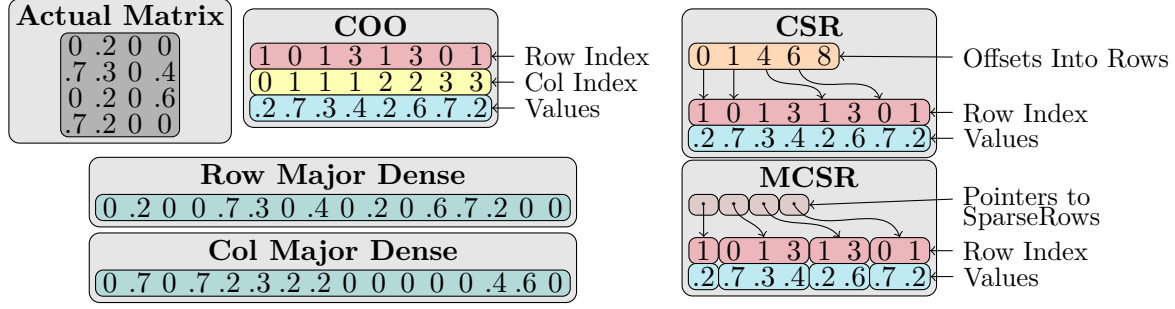


Figure 2.2: Basic Dense and Sparse allocations of a Matrix

2.1 Lossless Compression Techniques

The section focuses on lossless data representations, starting with uncompressed formats. We focus primarily on techniques that allow direct access to underlying unique values, considering the lossless compression techniques as data structures surrounding values.

Notation: We use the following notation: \mathbf{A} is a matrix of n rows and m columns. A specific cell is $\mathbf{A}_{i,j}$, while a row is $\mathbf{A}_{i,:}$ and a column $\mathbf{A}_{:,j}$. The set of non-zero cells in \mathbf{A} is given by $\mathbf{A}_{\neq 0}$. The non-zero cells in row and column are $\mathbf{A}_{i,\neq 0}$ and $\mathbf{A}_{\neq 0,j}$. If the i -th row, is empty, we use $\mathbf{A}_{i,\forall 0}$ formally defined as $\forall j, \mathbf{A}_{i,j} = 0$, while if it contains at least one non-zero $\mathbf{A}_{i,\exists x}$ where $x \neq 0$.

The Dense Baseline: In many systems, dense vectors, matrices, or tensors encode into a single continuous array. For matrices, the single array typically is in a row-major or column-major format, as shown in Figure 2.2. Row-major allocations map values from rows into the array, following $i \cdot m + j$. While column-major oppositely assigns columns via $i + j \cdot n$. When traversing the structure, it is beneficial for cache efficiency to process in the same major direction. The memory size of these dense arrays scales linearly with the number of cells $\mathcal{O}(nm)$. A dense double matrix therefore scales in bytes according to $8nm$.

Blocking: A common limit of the number of cells in a matrix is $2^{31} - 1$, the limit of, for instance, TensorFlow that uses 32-bit signed integers to index elements. Java-based programs similarly cannot allocate basic arrays larger than unsigned integers. The limitation can be overcome by allocating more arrays, with two nested arrays for matrices, that slice a matrix up into different numbers of rows and columns that can be processed—and (de)compressed—independently through an indirection.

Sparse Layouts: Sparsity exploitation allows processing using only non-zero values. Figure 2.2 shows three common layouts COO, CSR and MCSR [201, 117, 175, 215, 125]. COO stands for *COOrdinate* and allocates additional index structures that encode coordinates of non-zero values. The coordinate format typically uses tuples of coordinates in a single array, with pairs of row-column coordinates, an approach used in TensorFlow [92], or in two separate row index and column index arrays, which we use in SystemDS. The allocation of COO scales linearly with non-zeros, $\mathcal{O}(\mathbf{A}_{\neq 0})$. However, each value costs more with two offsets and one value. For a COO double matrix, it translates to 16 bytes per value with two integers and one double. CSR stand for *Compressed Sparse Rows* and has an identical row index array to COO. However, instead of allocating column offsets, it allocates a list of indexes that *point to rows* (ptr), indicating the start of row indexes of individual rows. CSR scale according to $\mathcal{O}(\mathbf{A}_{\neq 0} + n)$, with a fixed cost for each row in the ptr index. The *Compressed Sparse Columns* (CSC) scheme swaps ptr with pointers to column starts and row indexes with column. CSC has a fixed added cost in the number of columns, $\mathcal{O}(\mathbf{A}_{\neq 0} + m)$. For a double matrix, each non-zero in CSR or CSC costs 12 bytes (one integer and one double). A downside to CSR is that ptr and row index arrays reallocate if we want to add a single non-zero value to a row. DCSR [46] is a technique to avoid the extra overhead of rows with no non-zeros by adding an extra array containing row numbers with non-zero values. MCSR is a modifiable CSR representation that allows adding and removing values to the CSR compression without

reallocation. MCSR solves this by independently allocating row indexes and values in different arrays. Each row can then resize on demand in MCSR. With these techniques, the minimum sparsity required for memory improvements is 50% nnz with COO and 66.6% nnz with CSR or CSC.

Dense Dictionary Coding (DDC): Dictionary encoding [74, 168, 240, 18, 130, 124] compresses data by exploiting the number of unique elements, d . A column-major DDC encoding contains two parts: a *dictionary* with the d value tuples $\in \mathbf{A}_{:,i}$ (shown in Figure 3.2 as a dictionary with three tuples), and an *index structure* with a row-to-tuple mapping (e.g., dictionary position). DDC is dense because each row input is assigned a code in the map, making the length of the mapping array always equal to the number of rows. The mapping array must use a bit-width $\#B = \lceil \log_2(d) \rceil$ that can encode integers $\geq d$. The in-memory size of DDC scales

according to $\mathcal{O}(dm + \#Bn)$ where dm is the number of values contained in the dictionary multiplied by the number of columns, and $\#Bn$ is the number of rows multiplied by the used bit-width. If the underlying values are double, then the DDC in Figure 3.2 can be encoded with two bits bit-width because $d = 3$, resulting in an encoded size of $2 \cdot 10\text{bits} + 8 \cdot 3 \cdot 2\text{byte}$ rounded up to nearest byte 51 bytes, much smaller than the dense $10 \cdot 2 \cdot 8 = 160$ byte.

CSR-VI: CSR-VI [130] is a compound technique that uses dictionary encoding on the values of a CSR compressed representation. CSR-VI is the first of many examples of leveraging the compound effects of two techniques on top of each other. CSR-VI uses a DDC scheme that only supports encoding single values (not tuples). The DDC encoding in CSR-VI, therefore, is neither row nor column-oriented. It instead maps each non-zero value. CSR-VI scale according to $d + \#B \cdot \mathbf{A}_{\neq 0} + n$, which reduces to $\mathcal{O}(\mathbf{A}_{\neq 0} + n)$ because $d \leq \mathbf{A}_{\neq 0}$ is guaranteed. In the example, assuming integers for offsets and pointers and doubles for the dictionary, the size becomes $4 \cdot 2 + 4 \cdot 5$ bytes for the CSR part, and $2 \cdot 5\text{bits} + 8 \cdot 3\text{bytes}$ for the DDC part, totalling 54 bytes, slightly smaller than pure CSR 64 bytes, and much better than dense 160.

Frame of Reference (FOR): FOR use reference values to offset encoded values [88]. FOR does not improve compression alone. Instead, it changes the data into forms amenable to other compression techniques. FOR can, for instance, be applied to a full matrix to make it sparse by subtracting the most common values to keep as reference values shown in Figure 2.5. The value matrix can then recursively compress into a sparse format such as CSR. Another variation of FOR, let's call it FORL, uses the minimum value of columns as reference [264]. FORL can compress if the value inputs are $\in \mathbb{N}$ as integers, then each \mathbb{N} can be encoded with

$\#B = \lceil \log_2(\max(\mathbf{A}) - \min(\mathbf{A}) + 1) \rceil$. On the other hand, if the values encoded are $\in \mathbb{R}$ as floats, then values have to be recursively compressed via other compression schemes to result in compression. Accidentally, FORL is equivalent to FOR in Figure 2.5. A downside of both types of FOR in the context of compressed linear algebra is that it loses direct access to underlying data, where operations that cannot apply directly to the reference values must resolve the difference between reference and underlying values to get actual cell values.

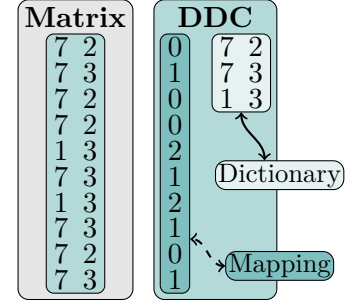


Figure 2.3: DDC scheme

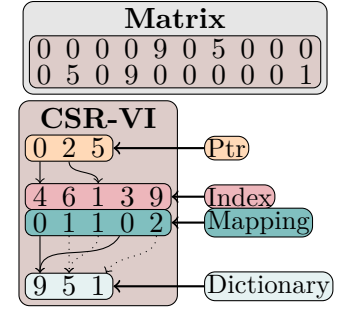


Figure 2.4: CSR-VI scheme

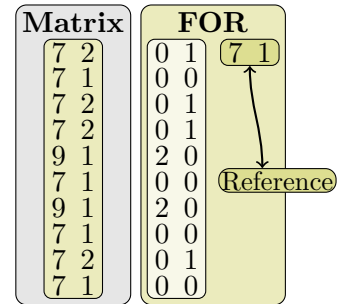


Figure 2.5: FOR scheme

Offset-list Encoding (OLE): In its purest form, OLE [74] encodes the offsets into a matrix that contains a specific value. It consists of the three parts shown in Figure 2.6: A dictionary of unique tuples, an offset list containing offsets of individual tuples, and a pointer list to indicate each tuples start in the offsets. This encoding is similar to CSR and is also in the context of the CLA papers [74], a sparse encoding that skips all zero tuples. The scaling of OLE is $\mathcal{O}(dm + z)$, where dm is the dictionary cost, and $z = (\mathbf{A}, \exists x \neq 0)$ is the number of tuples with at least one non-zero. CLA extends the basic OLE scheme by encoding offsets from row block segments of 2^{16} to encode each offset with only two bytes. The blocking makes the asymptotic scaling worse $\mathcal{O}(dm + z + db)$, where db is the cost of the offset pointers allocated for each unique value for each block segment. On one side, the blocking incurs an overhead in OLE that is significant only in ultra-sparse cases where there $z = 0$ in blocks. On the other side, the blocking improves operation performance, allowing threads to process different blocks cache consciously and independently.

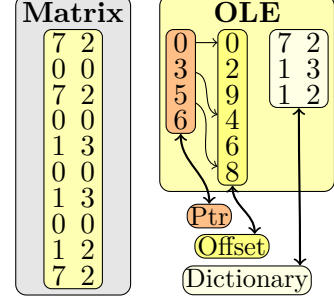


Figure 2.6: OLE scheme

Run-length Encoding (RLE): Long sequences of the same value commonly occur, and run-length encoding exploits these repeated values. There are many versions of RLE, many shown in Figure 2.7. (1) Independent RLE, I-RLE, where each run is encoded together with its value. I-RLE encodes a list of run lengths and a value array of associated tuples and scales according to $\mathcal{O}(mr)$ where r is the number of runs. However, it is common to combine RLE with a DDC encoding, DDCRLE, also shown in Figure 2.7. The downside of the DDCRLE scheme is the added indirection of the mapping. However, its memory consumption scales better, $\mathcal{O}(r + dm)$, assuming a low number of distinct tuples. CLA [74, 75, 73] choose to opt for a different flavour of RLE encoding, shown as RLE, that avoids the indirection of a DDC map but still allows storing unique tuples in a dictionary. It consists of three parts the dictionary of unique values, a pointer array (in orange), and an offset/run-length array (in green). The pointer array indicates the start and end positions in the run array of each unique tuple in the dictionary. In Figure 2.7, there are two runs for the first tuple and one run for the second. The run array works in pairs of two values. The first indicates the offset from the last run’s end to the start of the next run. The second contains the run’s length. The RLE encoding is further specialised in sparse processing because the schema skips encoding zero tuples. Finally, on top of everything else CLA improve RLE by applying it—similar to OLE—in blocks of 2^{16} making the overall scaling $\mathcal{O}(dm + r + db)$ where b is the number of blocks. While the RLE in CLA [74] encoding is a bit complicated, it makes up for it in compressed operations.

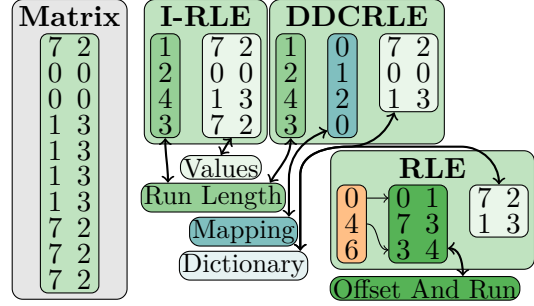


Figure 2.7: RLE scheme

Constant Encodings: While trivial, it is common to encounter blocks or sometimes even entire columns that contain a single value. We, for instance, observe constant columns in algorithms that fit an intercept value. Algorithms can, as a preprocessing step, append a column of ones to the feature matrix and, without any other changes, fit the intercept values implicitly. Another instance is in dense layers of neural networks, where instead of adding a bias after matrix multiplication, we can add a column of ones and an extra row of the weight matrix containing the bias values. These approaches are correct because of equivalent results with appended ones as shown in Equation (2.1). Constant encodings are small and scale according to $\mathcal{O}(dm)$. CLA [74] do not use constant encodings and instead rely on RLE.

$$AW + b = \begin{bmatrix} A & o \end{bmatrix} \begin{bmatrix} W \\ b \end{bmatrix} \quad (2.1)$$

Delta Encoding: In many ways similar to FOR, delta compression [124] encodes values as deltas from previous values. Most commonly, it is deltas from the exact previous value. However, another variation applies to windows of data and encodes values relative to the first value in a window. Figure 2.8 shows an example of delta encoding. Also, similar

Matrix	Delta
5 0	5 0
6 2	1 2
7 4	1 2
8 8	1 2

Figure 2.8: Delta scheme

to FOR, delta encoding does not improve compression size. Instead, it relies on recursively applying other compression techniques to its encoded output. Delta encoding is very efficient in columns with auto-incrementing values because the delta between all rows is equivalent. In such cases, a column can go from $d = n$ to $d = 1$, making the delta-encoded values perfect for RLE or even sometimes constant compression. There are three major downsides to delta encoding in the compressed linear algebra context. First, we lose direct access to the underlying values due to indirect additions, invalidating performing non-linear operations directly on the encoded values. Second, collecting the statistics to analyze delta encoding efficiency is expensive because we must calculate the delta values. Third, we cannot estimate the delta encoded size with the same statistics as other compression techniques because the recursive compressions rely on the statistics from delta encoded values.

CSR-DU: From the same paper as CSR-VI, CSR-DU [130] for CSR Delta Unit, replace the CSR ptr and row offsets with a compressed byte array called *ctl*. CSR-DU combine elements from delta encoding and OLE. Without going into details, *ctl* contains units of information that delta encodes offsets between cell positions with non-zero values. It distinguishes itself from OLE by not using offsets from the beginning but instead deltas between values. CSR-DU is an interesting example of focusing on compressing the index part of a compressed format instead of focusing on compressing the values.

Patched: Another compression algorithm family is called patched compression. Patched compression is another compound technique fused with other compression schemes. The fused encoding names prepended a P on the different compression algorithms, such as PFOR, PFOR-DELTA, and PDICT [264]. Patch compression treats values as *coded* or *exceptions*. Coded values are compressed into underlying compression schemes, while exceptions are, as the name implies, handled extraordinarily in separate data structures and serve as corrections on decompression. PDICT, for instance, encodes a dictionary with the most common values to reduce the mapping size from DDC by keeping rare values separated from the DDC dictionary to reduce the $\#B$ used in the dictionary mapping. Patched compression contains a design decision on how the exceptions overwrite the compressed values. Two common approaches are to overwrite cells or use exceptions as reference values and add them into cells.

Null Suppression (NS): Null suppression [200, 55] is another technique with many variations. The basic idea is to exploit consecutive nulls/zeros of individual integer values and replace them with descriptions of how many zeros there were and where they are located [3]. There are two common approaches to null suppression: packed or individual. An individual encoding contains a prefix for every value, sometimes called the *group-variant* [55]. An example of this prefix is two bits that indicate if a following integer can be represented in one, two, three, or four bytes [3]. A packed encoding chooses a prefix for a block of values, e.g. 32, and determines a $\#B$ per block. The mapping part in DDC is a prime use case for null suppression techniques. However, null suppression makes variable-length encodings that require additional indirections to find specific cells. Therefore, we do not use it in this thesis.

Lempel Ziv (LZ): LZ77 [262] and LZ78 [263] are two dictionary inspired compression algorithms. LZ algorithms compress data by using *literals* and *copy* instructions. A literal contains raw copies of the input, while a copy contains an offset back to decompressed values and a length indicating how to append seen values. The copy instructions are analogous to dictionary entries, making the algorithm part of the family of dictionary encoders. To find equal previously seen data, default LZ compression techniques keep a fixed-length sliding window, w , to search for matches. The longer the window is, the bigger the compression potential becomes. However, window size and how diligently it is analyzed for matches dictates

2. Background

the computational load of LZ algorithms. An interesting property of LZ compression that is highly encouraged is to copy lengths that exceed the current position, e.g., copy instructions with two indexes back and copy ten values. Such a copy instruction implicitly replicates the last two elements five times. LZ is a base of well-known compression formats, including GIF, PNG, and ZIP. The best case compression for LZ algorithms happens for empty or constant files scaling $\Omega(1)$, closely followed by short repeated patterns that fit within the windows size $\mathcal{O}(w)$. While the worst case is data with no repeated patterns $\mathcal{O}(n)$.

Lempel Ziv Welch (LZW): LZW [239] is a lossless variable length compression technique that encodes and decodes data by rediscovering dictionary entries. LZW compression relies on a predetermined dictionary of all unique single tokens in a dataset. In byte aligned LZW compression the dictionary can be filled with all bytes possible (256 unique values), that does not need to be allocated. Figure 2.9 shows an example of encoding a sequence of tokens, with the LZW algorithm. The example starts with the base dictionary, of all unique tokens 0, 2, and 3. In each row of the encoding table, we first find the longest sequence in the dictionary of subsequent input tokens. Then, we extend the dictionary with the longest sequence found plus the next token, and output the current dictionary value. The algorithm loop until the end of the input tokens is reached. As an extra trick, LZW

Input Tokens			
0	2	0	0 2 3 0 2 0 2 0 0
Base Dictionary			
0	→	0	, 2 → 1 , 3 → 2
Encode			
Seq	Next	Extend	Out
0	2	[0, 2] → 3	0
2	0	[2, 0] → 4*	1
0	0	[0, 0] → 5	0
[0, 2]	3	[0, 2, 3] → 6	3
3	0	[3, 0] → 7	2
[0, 2]	0	[0, 2, 0] → 8*	3
[0, 2, 0]	0	[0, 2, 0, 0] → 9	8
0	E		0

Figure 2.9: LZW example

can choose to encode the outputs with the a number of bits according to the size of the dictionary, and each time the dictionary size grows above powers of two, the output encoding use one more bit. This change is shown in Figure 2.9 with *. LZW’s runtime is equal to the length of input tokens $\mathcal{O}(n)$, and its worst-case output encoding is also $\mathcal{O}(n)$ and in the best-case (constants) $\Omega(\log(n))$. However, LZW’s worst case memory usage scales linearly with input size for the temporary dictionary extensions.

Grammar-based Compression: Another family of compression techniques is grammar-based compression, which uses context-free grammar (CFG) to compress data. A CFG contains a set of rules of two classes: terminals, T and nonterminals, L . A single nonterminal rule decomposes into one or many other terminals and nonterminals, e.g. $L_1 \rightarrow L_2 T_1 T_2 L_2$. Terminals do not decompose and contain basic blocks of data. The best possible grammar of any sequence, length n , is guaranteed of $\mathcal{O}(n/\log_2 n)$ [77]. However, finding the smallest grammar is NP-hard [77], therefore most grammar finding algorithms are greedy algorithms [77].

Entropy Encoding: Another class of compression techniques are entropy encoders. They adapt to the probability distribution of distinct values. According to Shannon’s theorem [211], the theoretical optimum is $-\log_2 P_i$ bits for each value in randomly ordered data, where P_i is the probability of a distinct value d_i . If all unique values are equally likely, and d is a power of two, DDC achieves this optimal compression using the same number of bits for each d . However, equal probabilities are rarely the case. If a compression technique adheres to Shannon’s theorem, it will scale according to $\mathcal{O}(\sum_{i=1}^d -\log_2 P_i \mathcal{F}_i)$, where \mathcal{F}_i is the frequency d_i . We mention three entropy encoding techniques Huffman, Arithmetic, and ANS.

Huffman: Huffman encoding is a entropy encoding technique invented by David Huffman in 1952 [111] that builds a prefix tree based on all distinct values and their frequencies. The prefix tree encodes meaning in the traversal of the tree. The Huffman encoding tree’s nodes always contain two child nodes, where traversing edges towards child nodes corresponds to setting a bit to zero or one. Meanwhile, leaf nodes are associated with distinct values. Values are decoded by traversing the prefix tree from the

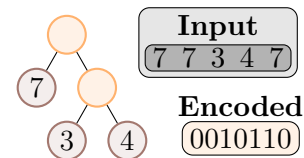


Figure 2.10: Huffman scheme

root node to the leaves, and similarly, values are encoded by traversing from value leaves to the root (and reversing the result). The Huffman encoding algorithm constructs the trees by minimizing the number of edges used for a given the frequency distribution of values. Morse code might seem similar to a Huffman encoding. However, it is not because Morse code relies on three signals. If we construct a Morse code tree with only dashes and dots, then some internal nodes would correspond to a letter, breaking Huffman coding that requires leaves to be the only source of distinct values. Because traversing the tree use different numbers of edges, Huffman encoding is a variable length encoding scheme. Huffman encoding can achieve Shannon’s optimum compression in cases where all properibilities are $P_i \in 2^{-\zeta}$ where $\zeta > 0$ and $\zeta \in \mathbb{N}$. However, because Huffman encoding is a variable-length encoding scheme, decoding can only happen from the beginning of compression blocks, and random access to specific indexes of values is not permitted.

Arithmetic coding: Witten et al. [244] (1987) states that ‘Arithmetic coding is superior in most respects to Huffman’. While Huffman encoding returns a stream of bits corresponding to edges in the prefix tree, arithmetic coding returns a continuously refined infinite precision \mathbb{R} value. Arithmetic coding allows $\zeta \in \mathbb{R}$, encoding each value without being bit-aligned. If the frequencies of all values are powers of two, then Huffman encoding produces the same compressed size. However, when this is not the case, Arithmetic compression produces smaller encodings. Appendix A.1 shows an example of arithmetic encoding. While the compressed result from Arithmetic compression is small, it is even more computationally expensive than Huffman encoding. Witten et al. [244] point out that LZ-like techniques are more appropriate when the aim is raw speed. Unfortunately, arithmetic encoding, just like Huffman, requires sequential decompression, making indexed lookups prohibitively expensive.

Asymmetric Numeral Systems (ANS): The final lossless technique presented, and also the newest in this list from 2013, is Asymmetric Numerical Systems [72]. ANS compress data into a single (also infinitely scaling) natural number N by continuously appending bits to the output. Instead of using the actual probability distributions like Arithmetic coding, ANS use an approximation of the probabilities P_i into a state space L . A large state space L approximated the probabilities more precisely, giving better compressions. There are three variants of ANS binary (uABS), range (rANS), and tabled (tANS). uBAS encode only bits, always reducing the number of unique values to two. rANS approximate P using ranges and limits memory usage if many distinct values are encoded. However, it adds arithmetic work for compression and decompression compared to tANS. Finally, tANS approximate P using a state space L with $|L|$ cells, where each d_i gets assigned P_i fraction of cells depending on their relative probability. The state space in tANS enables a finite-state machine construction that, via lookups, enables fast compression. Interestingly, ANS encode and decode in opposite directions, meaning if compressed from the beginning, it decompresses from the end.

Limitation: While all the compression techniques mentioned so far can compress data, there is no guarantee that they will compress. It is quite the opposite, which can be proven via the pigeonhole principle [145] (also called Dirichlet’s drawer principle [69]). The principle states that it is impossible to put m pigeons into n containers when $m > n$ without at least two pigeons in one container. Therefore, let’s say we have a lossless compression technique and its associated decompression algorithm. Because the compression technique is lossless, any compressed state has only one decompressed result. Then, let’s imagine we are given all permutations of k bits (equal to 2^k pigeons). Compression implies a reduction in size $< k$. All permutations can’t compress because some inputs would compress to the same state. In the pigeonhole metaphor, there would be more than one pigeon in a container.

Compression Algorithms: While we covered the end compressed format of various techniques, there are many algorithms specializing in how to compress efficiently. The specifics of these algorithms are out of scope in this background.

2.2 Categorization of Compression Techniques

The introduced compression techniques fall into different categories. While all are lossless, some are more computationally expensive than others or contain limitations that would impact the performance of compressed linear algebra. We classify these into two categories.

Heavyweight: The heavyweight techniques include the entropy encoders such as Arithmetic [244], Huffman [111], and ANS [72]. The entropy encoders are heavyweight because of two properties. First, they introduce a non-trivial overhead in decoding individual values, and they require sequential and directional decompression, thereby limiting their flexibility for our use cases. LZ(W) and grammar-based compression are also heavyweight in previous work on CLA [74]. However, recent work shows promise in fusing LZ or grammar-based compression with CLA [147, 146, 77]. Related work also classify these techniques as heavyweight [134, 55].

Lightweight: On the other side, lightweight techniques include Sparse techniques, DDC, FOR, OLE, RLE, Delta, NS, and Constants. These techniques are arguably simpler, and many cases have low overhead in lookups in their compressed structures. Furthermore, as a bonus, most do not require sequential scanning or decompression.

2.3 Compression Systems / Compound Techniques

Only using one compression technique is insufficient to cover many different types of redundancy in data. Therefore, systems often use compound compression techniques. A classic compound technique could be DEFLATE [68] that combines the LZ77 [262] variant with Huffman coding. DEFLATE is used in the GZIP, PNG, and ZIP formats. Some systems are classified as general compressors such as GZIP, Snappy [90] and ZStd [76], while others choose to specialize in specific modalities such as text, images, or sound. Examples of specialized compression could be PNG, MP3 or MP4. We focus on frameworks for "high speed and reasonable compression" [90].

Snappy: Snappy is a compression framework from Google using an LZ77 stream byte-oriented encoding that compresses blocks in a single pass [90, 89]. Snappy stores *literals*, which are uncompressed data, and *copies*, which are references back into previous decompressed data. The biggest difference Snappy has compared to LZ77, is the offsets are indicated with 1, 2 or 3-byte offsets.

ZStd: ZStd [76, 51], developed by Meta, compresses one or more *frames*. Each frame can be independently compressed and decompressed. A frame contains one or more blocks that use three types of block encoding. 1 raw, 2 RLE, and 3 compressed. Raw is uncompressed data, RLE is a single byte of data and a byte for how long a run is, and compressed consists of two sections *literals* and *sequences*. *Literals* are stored uncompressed with Huffman codes, RLE, ANS, or others [51]. *Sequences* are copy commands, specifying offset and copy lengths. ZStd contains many more minute details and recursive encodings, found in their RFC specification [51]. While not stated in the specification, the ZStd is heavily inspired by LZ77 fused with RLE, ANS, and Huffman encoding.

Framework Users: Many systems do not implement their own compression encodings, instead relying on other compressors. Spark [254] 3.x uses LZ4—another high-speed LZ77 variant—per default but has support for Snappy and ZStd. Flink [47, 16] defaults to Snappy for internal compression. TensorFlow [158], interestingly default to ZLIB or GZIP for their TFRecords. Scientific data formats like NetCDF and HDF5 [97] and storage managers like SciDB [217] and TileDB [180] also use other libraries' compression.

General Compression Random Access: A problem for LZ77, LZW, Snappy, GZIP, and ZStd is that encoded data does not allow random access to the uncompressed data, and therefore their techniques do not allow processing directly on their compressed formats. While total random access is not strictly required because it is possible to scan some elements to recover the given data, it does introduce overheads.

Table 2.1: Linear Algebra Asymptotic Runtimes. \mathbf{A} is input, \mathbf{R} is the dense output, c is a constant, \mathbf{v} is a vector, \mathbf{M} is a dense matrix, and $@$ is matrix multiply.

Algebra Type	sum(A)			max(A)			A + <i>x</i>			A · <i>x</i>		
	all	col	row	all	row	col	c	v	M	c	v	M
Dense	$\mathcal{O}(nm)$			$\mathcal{O}(nm)$			$\mathcal{O}(nm)$			$\mathcal{O}(nm)$		
COO	$\mathcal{O}(\mathbf{A}_{\neq 0})$	$\mathcal{O}(\mathbf{R} + \mathbf{A}_{\neq 0})$		$\mathcal{O}(\mathbf{A}_{\neq 0})$	$\mathcal{O}(\mathbf{R} + \mathbf{A}_{\neq 0})$		$\mathcal{O}(nm)$			$\mathcal{O}(\mathbf{A}_{\neq 0})$		
CSR	$\mathcal{O}(\mathbf{A}_{\neq 0})$	$\mathcal{O}(\mathbf{R} + n + \mathbf{A}_{\neq 0})$		$\mathcal{O}(\mathbf{A}_{\neq 0})$	$\mathcal{O}(\mathbf{R} + n + \mathbf{A}_{\neq 0})$		$\mathcal{O}(nm)$			$\mathcal{O}(n + \mathbf{A}_{\neq 0})$		
CSR-VI	$\mathcal{O}(\mathbf{A}_{\neq 0})$	$\mathcal{O}(\mathbf{R} + n + \mathbf{A}_{\neq 0})$		$\mathcal{O}(d)$	$\mathcal{O}(\mathbf{R} + n + \mathbf{A}_{\neq 0})$		$\mathcal{O}(nm)$			$\mathcal{O}(d)$	$\mathcal{O}(n + \mathbf{A}_{\neq 0})$	
Algebra BLAS	A@v	v@A	A@M				M@A				A^T@A	
	GEMV	GEVM	GEMM								SYRK	
Dense	$\mathcal{O}(nm)$		$\mathcal{O}(nmk)$						$\mathcal{O}(nm^2)$			
COO	$\mathcal{O}(\mathbf{R} + \mathbf{A}_{\neq 0})$		$\mathcal{O}\left(\mathbf{R} + \sum_{i=1}^n \mathbf{A}_{i,\neq 0}k\right)$		$\mathcal{O}\left(\mathbf{R} + \sum_{i=j}^m \mathbf{A}_{\neq 0,j}k\right)$		$\mathcal{O}\left(\mathbf{R} + \sum_{i=1}^n (\mathbf{A}_{i,\neq 0})^2\right)$					
CSR(-VI)	$\mathcal{O}(n + \mathbf{A}_{\neq 0})$		$\mathcal{O}\left(\mathbf{R} + n + \sum_{i=1}^n \mathbf{A}_{i,\neq 0}k\right)$		$\mathcal{O}\left(\mathbf{R} + n + \sum_{i=j}^m \mathbf{A}_{\neq 0,j}k\right)$		$\mathcal{O}\left(\mathbf{R} + n + \sum_{i=1}^n (\mathbf{A}_{i,\neq 0})^2\right)$					

2.4 Compressed Linear Algebra

This section describes compressed linear algebra (CLA) frameworks that use compressed formats and perform linear algebra directly on their compressed formats. The section assumes operations are performed on floating point data, and we focus on the limited set of key operations. We start with a baseline of dense operations.

Comparison: Table 2.1 shows the asymptotic runtime of common operations in Linear Algebra. The dense operations’ asymptotic runtime is based on the dimensionality of the input. It is hard to directly analyze performance via big-O notation, and therefore, we need to take number of index lookups together with cache-locality into account for a thorough analysis. To lighten the scope of the background, we put much of the detailed analysis in Appendix A.2.

Dense Linear Algebra: The dense baseline is most commonly used since it has stable performance with different data characteristics. Basic Linear Algebra Subprograms (BLAS) libraries are the de-facto standard for performing dense linear algebra [140, 29]. Example libraries include cuBLAS [174] for NVIDIA GPUs, Intel MKL [117] (now under oneAPI [80]) and OpenBLAS [179] for CPU. While a high degree of freedom is given to people implementing the BLAS algorithms, to allow for e.g. Strassen’s algorithm [220], most implementations have the same underlying asymptotic runtime.

Sparse Linear Algebra: The beauty of sparse linear algebra is that most operations can skip zero values in the matrices. Skipping non-zero values changes many operations’ asymptotic behavior, making sparse operations preferable once the overhead of its added index structures is less than the asymptotic gains. The break-even point highly depends on the sparse format selected and the operations performed (See Appendix A.2 for details). A key takeaway is the difference between COO and CSR is the guaranteed additional cost of n to scan all non-zero values because CSR forces processing each row in many cases. However, once a non-trivial number of non-zeros are contained in $\mathbf{A}_{\neq 0}$, operations load fewer values per operation because CSR implicitly materializes row indexes while iterating through the data structure.

CSR-VI: We included CSR-VI [130] in the table to highlight one of the key motivations for CLA [74]. CSR-VI focused on $\mathbf{A}@\mathbf{v}$ and $\mathbf{v}@\mathbf{A}$, showing that if the number of unique values in a sparse representation is low, it can improve CSR performance via better cache locality using the smaller CSR-VI layout. While CSR-VI does not change the asymptotic behavior of the vector multiplication, the reduced amount of values loaded improves performance because vector multiplication is memory bandwidth bound. CSR-VI could enhance the performance of many other operations for the same reason. However, notably marked in red, CSR-VI reduces $\mathbf{A} \cdot c$ ’s asymptotic behavior to scale in the number of unique values, d . Appendix A.3 contains more in-depth details on why CSR-VI improve performance. The improvement motivates exploration into compression-based enhancement of linear algebra.

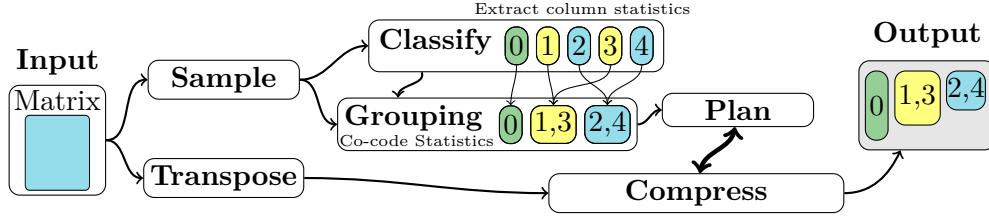


Figure 2.11: CLA Compression Sequence.

2.4.1 CLA in SystemML

CLA [74] is fully integrated into SystemML [86, 32, 33]. CLA’s main use case was to fit data in the total aggregated memory of a Spark cluster to avoid spilling data partitions to disk. CLA in SystemML combines various compression techniques to encode columns and column groups into lightweight compressed formats designed for cache-conscious operations without decompression. Importantly, CLA returns the same results as if the data was uncompressed, making it possible for the optimizing compiler to inject compression into a compiled program without user input. Another goal of CLA was to have close to (or better) uncompressed in-memory performance. Therefore, CLA was only enabled per default if the matrices for a workload exceeded aggregate cluster memory.

Compressed Format: CLA use four different compression schemes: DDC, RLE, OLE, and UC (uncompressed). The compression schemes are used on subsets of columns, called column groups (not to be confused with column group statistics [99]), and each group contains the encoding and an array of column indexes for that group. The memory usage of each group is equal to the previous section, with the addition of maintaining a column mapping. The many schemes allow CLA to adapt to the characteristics of underlying data. The CLA paper’s solution struggles in two scenarios. First, ultra-sparse matrices where the compressed formats do not beat sparse formats. Second, large numbers of columns. CLA’s compression is comprised of a few stages shown in Figure 2.11.

Compression Planning: The first task is to decide on a compression plan. CLA first collect statistics of individual columns. Second, which columns to combine into multi-column encodings. The grouping of columns is called co-coding. Finally, CLA chose the compression scheme for each group of columns. To keep the planning fast, CLA uses sampling-based techniques to collect statistics to estimate compression sizes. Deciding on columns to encode together can be expensive with exhaustive search $\mathcal{O}(m^m)$ and brute force greedy $\mathcal{O}(m^3)$. Instead, CLA employs a greedy algorithm with memoization of column groupings $\mathcal{O}(m^2)$, which is still prohibitively expensive for large numbers of columns. On top of this, CLA groups columns into different bins based on the cardinality of underlying groups and then runs the greedy co-coding algorithm in parallel on each bin. The end result is the compression plan.

Compression: Each column group is compressed in parallel. Compressing a single group first, collect statistics of the group’s columns from the input matrix. Then, an optimal encoding is chosen based on the collected statistics. When the compression plan is based on a sample, CLA can correct false positives from the planning phase and change the selected compression scheme, split column groups, or return UC groups while compressing groups. To enable cache-efficient scans of the input, CLA chose to transpose the input matrix before compression.

Distributed: CLA supports both local-in-memory compression of entire matrices and distributed compression. The distributed Spark implementation in SystemML uses resilient distributed datasets key-value pairs of row-column indices and matrix blocks. CLA independently compress each block of data. The downside to CLA’s distributed design is that redundancy cannot be exploited across blocks of data, and therefore, block sizes have to increase to improve compression potential. The default block sizes in SystemML and SystemDS are 1k blocks, while the CLA papers [74, 75] found 16k blocks to be a better size for compressed distributed blocks.

Table 2.2: Linear Algebra Asymptotic Runtimes of Aggregate and Binary Operations. \mathcal{G} is a column group, \mathcal{DC} stand for decompression, and other variables can be looked up in Table 2.1

Algebra Type	$\text{sum}(\mathcal{G})$			$\text{max}(\mathcal{G})$			$\mathcal{G} + x$			$\mathcal{G} \cdot x$		
	all	col	row	all	col	row	c	v	M	c	v	M
DDC	$\mathcal{O}(n + dm)$	$\mathcal{O}(nm)$		$\mathcal{O}(dm)$	$\mathcal{O}(nm)$		$\mathcal{O}(dm)$	\mathcal{DC}		$\mathcal{O}(dm)$	\mathcal{DC}	
OLE	$\mathcal{O}(b + z + dm)$	$\mathcal{O}(zm)$		$\mathcal{O}(dm)$	$\mathcal{O}(nm)$		$\mathcal{O}(n + dm)$	\mathcal{DC}		$\mathcal{O}(dm)$	\mathcal{DC}	
RLE	$\mathcal{O}(b + r + dm)$	$\mathcal{O}(zm)$		$\mathcal{O}(dm)$	$\mathcal{O}(nm)$		$\mathcal{O}(n + dm)$	\mathcal{DC}		$\mathcal{O}(dm)$	\mathcal{DC}	
UC	$\mathcal{O}(nm)$			$\mathcal{O}(nm)$			$\mathcal{O}(nm)$	\mathcal{DC}		$\mathcal{O}(nm)$	\mathcal{DC}	
Algebra	$\mathcal{G}@v$		$v@\mathcal{G}$	$\mathcal{G}@M$		$M@\mathcal{G}$	$\mathcal{G}^\top @\mathcal{G}$					
DDC	$\mathcal{O}(n + dm)$			\mathcal{DC} rows and use $v@\mathcal{G}$			\mathcal{DC} rows and use $v@\mathcal{G}$					
OLE / RLE	$\mathcal{O}(b + z + dm)$			\mathcal{DC} rows and use $v@\mathcal{G}$			\mathcal{DC} rows and use $v@\mathcal{G}$					
UC	$\mathcal{O}(nm)$			\mathcal{DC} rows and use $v@\mathcal{G}$			\mathcal{DC} rows and use $v@\mathcal{G}$					

Notation: Let \mathcal{G} be a compressed a column group of n rows and m columns, and \mathcal{G}_i is the i 'th group in a compressed matrix. If \mathcal{G} has a dictionary component, it is noted as \mathcal{D} , while its index component is \mathcal{I} . z again refer to the count of non-zero tuples in the group $(\mathbf{A}_{:, \exists x \neq 0})$, while d is the number of distinct tuples. For RLE r is still the number of runs in the compressed format and, for OLE and RLE, b is the number of blocks compressed. To reduce confusion, we evaluate the asymptotic cost from the perspective of individual column groups without output allocation. However, the overall cost is the sum of all groups in a compression.

Decompression Default: For CLA the basic philosophy is that if an operation was not supported in compressed space, CLA would fall back to decompressing the matrix, and performing the equivalent operation on the uncompressed matrix.

Performance: The same operations as in Table 2.1 is shown in Table 2.2 for the different encodings of CLA. We replaced all instances of decompressing operators with \mathcal{DC} . CLA always decompressed to a dense format. The overhead of decompression for all encodings scales at maximum linearly with the number of cells the group encode. The decompression cost is, therefore, $\mathcal{O}(nm)$ for DDC and UC, while $\mathcal{O}(b + mz)$ for OLE and RLE.

Summation: Starting with aggregate operations, we can observe that co-coded versions of column groups are better than the uncompressed versions, while single column groups generally have asymptotic runtime equal to uncompressed dense or sparse allocations. The downside is that co-coding columns increase d multiplicatively with uncorrelated columns $d_{ij} \leq d_i \cdot d_j$.

DDC improves the sum operations by iterating through its index structure, \mathcal{I} , calculating, \mathcal{F} , the frequency counts of each tuple in the dictionary, \mathcal{D} . Once \mathcal{F} is computed, DDC accumulates dictionary entries multiplied with their frequencies according to:

$$\text{sum}(\mathcal{G}) = \sum_{i=1}^d \mathcal{F}_i \cdot \sum_{j=1}^m \mathcal{D}_{i,j}$$

If $d > 32 \cdot 1024$, then DDC in CLA falls back to an $\mathcal{O}(nm)$ solution that looks up dictionary entries directly via \mathcal{I} for each cell. The fallback happens because the overhead of allocating the temporary \mathcal{F} array can be expensive.

OLE skips through its encoding and only processes non-zero tuples z and, similarly to DDC, leverage frequencies of distinct tuples but does not need to allocate \mathcal{F} since it can process each tuple in \mathcal{D} independently.

RLE similarly benefits from \mathcal{F} . However, RLE further improves, relative to OLE, processing by counting run lengths in the underlying data r , but RLE especially has trouble processing row sums. All column groups struggle with row sums and fall back to performance similar to uncompressed, where DDC forces dense processing, and OLE and RLE can use sparse.

2. Background

Maximum: Computing the max or min values in CLA is very fast because the compression maintains a dictionary of unique values. The encodings can — like CSV-VI — return the max very quickly for the entire compressed matrix. There is an overhead compared to CSV-VI when columns are co-coded because the dictionary can contain duplicate values. However, CLA can also return the max value quickly for individual columns, unlike CSV-VI, but the row aggregates scale in the input size because each decompressed cell is processed for row aggregates in CLA. CLA max-like operations do not change across encoding types.

Densifying: When performing densifying operations such as addition, CLA struggles.

Vector or Matrix: Adding a column vector, row vector, or a matrix CLA decompress the entire compressed input first and subsequently performs uncompressed dense linear algebra.

Constant: On the other side, CLA exploits constant additions. However, the different encodings process such operations differently. Constant addition performs best with less co-coding because the reduced co-coding introduces fewer duplicate values inside the dictionaries. DDC performs very well when adding a constant, only operating on the internal dictionary. RLE and OLE can also have the same runtime as DDC. However, only if they compressed inputs with no zero tuples. If not, the expensive n term is added *detect* and *allocate* the zero rows in their schemes. Specifically, the allocation is costly because it requires RLE or OLE to analyze their current index structure and allocate new extended versions. The densifying increases z to $z = n$, increasing the OLE and sometimes RLE runtime larger than DDC in many operations. r similarly increases by the number of previous zero runs. However, it is guaranteed $r \leq n$.

Sparse-safe: Sparse-safe constant operations have excellent scaling properties that benefit from the dictionaries like CSR-VI. *Vector and Matrix:* Unfortunately, the vector or matrix input also use decompression when the operation is sparse-safe in CLA. *Constant:* Unlike densifying operations, OLE and RLE do not need to change their index structures and, therefore, can use an equal implementation to DDC.

Matrix @ Vector: $\mathcal{G}@v$ is called right matrix multiplication (RMM). Figure 2.12 shows an example of CLA’s RMM with the DDC encoding from Figure 3.2 applied to columns 1 and 3. The RMM is a three-stage process. First, RMM slices out values from v corresponding to the column indexes of the column group \mathcal{G} . The temporary array, v' , size is of the number of columns of the column group, m . Second, using the dictionary of the compressed format, the pre-aggregate, \mathcal{P} is calculated $\mathcal{P} = \mathcal{D}@v'$, $\mathcal{O}(dm)$. Notice the pre-aggregate is computed via a standard matrix multiply that can use generic matrix multiplication libraries. The \mathcal{P} result is an array in the size of distinct values d . Third, the pre-aggregated results in \mathcal{P} are added into the output cells according to the index structure of the column group, \mathcal{I} . The running time of the last step depends on the encoding in CLA. The runtime of the last stage is $\mathcal{O}(n)$ for DDC while skipping zero tuples $\mathcal{O}(z + b)$ for RLE and OLE. For cache efficiency in the final stage, RLE and OLE use their blocking structures, while DDC naturally adds sequentially. When the compressed matrix uses multiple encodings, each encoding adds to the same result vector. The CLA operation is memory bandwidth bound, just like an uncompressed matrix-vector multiplication. However, the CLA implementation scales better with a high degree of co-coding of columns since co-coding reduces the number of passes the operation does on the result vector.

Parallelization: For parallel execution, RMM in CLA creates a set of tasks that each process the multiplication of a row block of the compressed input. Unfortunately, for CLA’s implementation steps, one and two (slicing and pre-aggregation) are duplicated for each thread. However, this overhead is a minor problem when $n \gg d$.

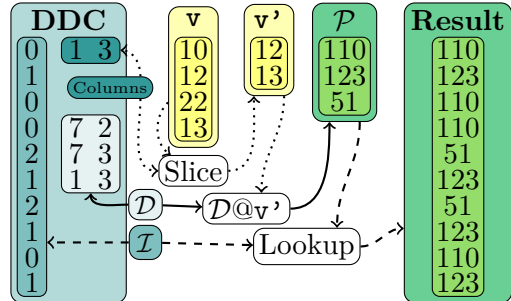


Figure 2.12: $\mathcal{G}@v$ DDC RMM Example.

Vector @ Matrix: $v@G$ in CLA called left matrix multiplication (LMM). LMM has two phases in CLA, which can be interpreted as a reversal of the opposite order of RMM. Figure 2.13 shows an example of LMM with a DDC column group. First, LLM computes a pre-aggregate \mathcal{P} . The pre-aggregation is different compared to RMM and follows the formula:

$$\mathcal{P}_j = \sum_{i \in \{i | \mathcal{I}_i = j\}} v_i$$

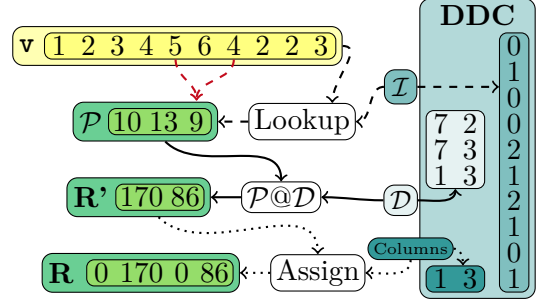


Figure 2.13: $v@G$ DDC LMM Example.

In essence, the pre-aggregate sum the values of \mathbf{v} by looking up the output index for \mathcal{P} in \mathcal{I} . Figure 2.13 shows, marked with red arrows, the calculation of $\mathcal{P}_2 = 5 + 4 = 9$. The runtime of LMM pre-aggregation depends on the compression type. DDC scale according to $\mathcal{O}(n)$, while RLE and OLE $\mathcal{O}(z + b)$. Second, LMM performs a matrix multiplication (called *postscaling* in the paper) with the column group dictionary \mathcal{D} and the pre-aggregate \mathcal{P} . CLA fused the multiplication with the assignment into the output cells of the result vector, avoiding a third phase. The runtime of the matrix multiplication and assigning into the output is $\mathcal{O}(dm)$. Similar to CLA’s RMM, LMM scales better with column groups containing many columns while having a small number of distinct tuples.

Parallelization: CLA naïvely parallelize over column groups for LLM. This choice is fine in cases with many column groups. However, it suffers when some groups are faster to process than others or when the number of column groups is less than the number of available threads.

Matrix @ Matrix: $M@G$ and $G@M$ were not directly supported in CLA. Instead, the operations slice out vectors from \mathbf{M} and use RMM or LMM depending on the side of the uncompressed input. This decision, unfortunately, kept the memory-bandwidth bound nature of matrix-vector multiplications but makes sense to reduce code.

Special cases: There are two special compound matrix multiplication operations that CLA specialized for. *MMChain* defined as $p = \mathbf{A}^\top @ (w \cdot (\mathbf{A} @ v))$ and *TSMM* $\mathbf{R} = \mathbf{A}^\top @ \mathbf{A}$. The *MMChain* is specialized because the intermediate result of $\mathbf{A} @ v$ would allocate a big vector intermediate in the size of n that after being multiplied with w (another vector in the number of rows) anyway would collapse into a small output vector of size m when matrix multiplied with \mathbf{A}^\top . The second operator $\mathbf{A}^\top @ \mathbf{A}$ exploits the symmetry of the result matrix, $\mathbf{R} = \mathbf{R}^\top$, and avoid calculating half of the result matrix. Both operations in CLA decompress columns one at a time and then leverage $G@v$ multiplications for each column group.

Tradeoff: A dilemma arises when choosing encodings and co-coding. For instance, the sparse-safe operations of CLA scale best with small dictionaries, while the aggregate sum and both matrix multiplication types scale better with more columns in each column group (assuming a low number of distinct values). CLA does not address this tradeoff because it fully optimize for compression size. CLA does mention *global planning* as future work [75]. The idea of *global planning* is to inform the compression to select an optimized compression scheme based on a collection of workload characteristics to be executed on the compressed matrix.

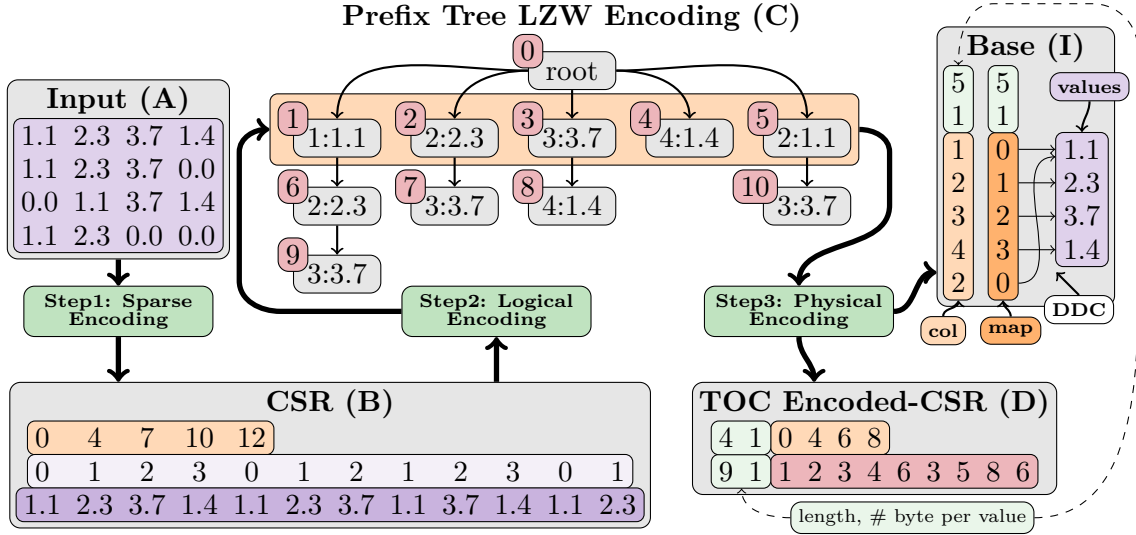


Figure 2.14: TOC Compression Example (modified from [147] to fit compression terminology).

2.4.2 Tuple-oriented Compression

Another influential work on compressed operations is Tuple-oriented Compression (TOC) [147, 146]. Unlike CLA, TOC uses a row compression scheme. TOC is designed for mini-batch stochastic gradient descent (MGD) and, therefore, aims to compress individual mini-batches such that entire datasets can fit in memory instead of reading mini-batches from disk.

Compressed Format: TOC divides its compression into three parts. (1) sparse encoding, (2) logical encoding and (3) physical encoding as shown in Figure 2.14. The first stage’s sparse encoding uses CSR to construct tuples of offset value pairs. The second uses a LZW [239] variant that encodes full tuples as LZW tokens from the CSR representation. The initial dictionary highlighted in the first row of the prefix tree of the LZW is all possible unique offset value pairs. In the paper, the authors call the LZW dictionary a prefix tree, which is common terminology for the same thing, just allocated in a tree-like structure in memory. Let \mathbf{C} be the fully-constructed prefix tree. Stage three encodes the physical encoding into two parts. First, the base dictionary, \mathbf{I} , and second, the encoded CSR table, \mathbf{D} . The base dictionary is recursively compressed with a DDC encoding on the values as the final compression. The encoded CSR table contains two arrays: an offset list for encoded row starts and an encoded array. Finally, the physical encoding prepends a tuple for all integer-based arrays to indicate the length and number of bytes used for each value.

Compressed Size: An estimate of a TOC compressed matrix size can be determined by combining the asymptotic growth of its compression components. The LZW component of TOC compresses each CSR row partially but with a shared growing prefix tree. Therefore, TOC wants to find similar rows with the same values in equivalent column indexes. Assuming the number of non-zeros is > 0 for each row, and each row contains equivalent values, then the LZW component would scale according to $\Omega(n + c)$, where c highest number of non-zeros on a row. However, the size would tend towards $\Omega(n)$ since new rows contain no new sequences. If each row contains either different column indexes or values, the scaling would be $\mathcal{O}(n + \mathbf{A}_{\neq 0})$. The combination of lower and upper bound scaling makes TOC appealing because it guarantees equivalent asymptotic scaling to CSR techniques, making it easy to estimate compression memory overheads. The extra recursive compression and packing in the physical design do not change the asymptotic scaling of the compression technique. CLA assumes the number of rows in the input data is large, while TOC targets mini-batches with few rows. Therefore, it makes sense for the TOC compression algorithm to exploit redundancy inside rows rather than CLA’s exploitation of columns.

Table 2.3: Linear Algebra Asymptotic Runtimes of Aggregate and Binary Operations in TOC. Missing operations compared to Table 2.1 and Table 2.2, are not supported.

Algebra Type	$\mathbf{A} + \mathbf{x}$		$\mathbf{A} \cdot \mathbf{c}$	$\mathbf{A} @ \mathbf{x}$		$\mathbf{x} @ \mathbf{A}$	
	\mathbf{c}	\mathbf{M}	\mathbf{c}	\mathbf{v}	\mathbf{M}	\mathbf{v}	\mathbf{M}
TOC	$\mathcal{O}(nm)$		$\mathcal{O}(d)$	$\mathcal{O}((\mathbf{D} + \mathbf{C}')k)$		$\mathcal{O}((\mathbf{D} + \mathbf{C}')k)$	

TOC supports four types of operations. These operations enable the execution of general linear models and neural network workloads.

Sparse-safe Scalar: The $\mathbf{A} \cdot \mathbf{c}$, together with \mathbf{A}^c , are sparse-safe operations, which is the fastest type of operation on TOC. The DDC encoding inside TOC isolates all unique possible values of the input. Therefore, it is possible to simply modify the dictionary to get excellent performance benefits from a global DDC encoding $\mathcal{O}(d)$. The benefit is equivalent to CSR-VI and better than CLA, which only exploits the distinct *tuples* inside individual column groups.

Unsafe Scalar: TOC rely on the sparsity exploiting CSR format. Therefore, they suffer from sparse-unsafe operations, like the OLE and RLE encoding in CLA. The densification leads to decompressing the entire encoding, $\mathcal{O}(nm)$ because the compression structure would change based on the newly introduced values in the CSR base of TOC. To explain the details, we first describe how decompression works in TOC.

Decompression: The process of decompressing a TOC encoded mini-batch is: First, build an index for the tree (\mathbf{C}) called \mathbf{C}' , as shown in Figure 2.15, to be able to efficient lookup the sequence of tuples for each code $\mathcal{O}(|\mathbf{C}'|)$. Note the index in Figure 2.15 is not allocated, while the col, value, and parents are. \mathbf{C}' is constructed via the decompression algorithm of LZW by unpacking the default dictionary to the first indices and scanning \mathbf{D} . Using \mathbf{C}' , the TOC-encoded CSR (\mathbf{D}) can be transformed to a CSR representation with $\mathcal{O}(\mathbf{A}_{\neq 0})$ time, followed by a decoding of the CSR to a dense matrix $\mathcal{O}(nm)$.

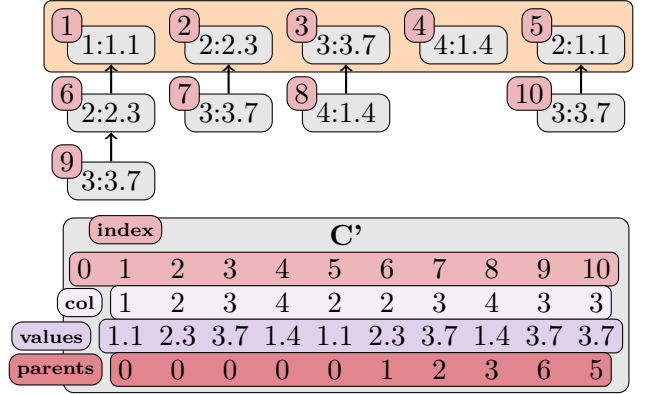


Figure 2.15: TOC Create \mathbf{C}'

RMM: Right matrix multiplication, $\mathbf{A} @ \mathbf{M}$, in TOC scale in the number of encoded tokens. TOC simplifies the matrix-matrix operation to a matrix-vector operation, $\mathbf{A} @ \mathbf{v}$, to process each column of the right-side matrix. In each $\mathbf{A} @ \mathbf{v}$ allocate a vector h of length $|\mathbf{C}'| + 1$ and calculate for each entry in \mathbf{C}' its multiplication value:

$$h_i = \mathbf{C}'_{i.val} \cdot v_{\mathbf{C}'_{i.col}} + h_{\mathbf{C}'_{i.parent}}$$

h_0 is always zero, while h_1 is the first rule. The nice characteristic of h is that the parent is already calculated when iterating from the beginning of \mathbf{C}' . After computing h , TOC iterates through \mathbf{D} to add into individual output cells by looking up indexes in h .

LMM: Left matrix multiplication, $\mathbf{M} @ \mathbf{A}$, is done in the opposite order of RMM. In LMM another, h is used. However, this one is in the size of $|\mathbf{D}|$. For each row \mathbf{v} in \mathbf{M} , we go through \mathbf{D} and add v_i to $h_{\mathbf{D}_{ij}}$. Similar to CLA's LMM pre-aggregation, TOC sums the values of the left side row-vector into h based on the indexes in \mathbf{D} . Following the summation, TOC loop *backwards* ($i = |\mathbf{D}|, \dots, i = 1$) though h and assign the output row R 's values to $\mathbf{R}_{\mathbf{C}'_{i.col}} += h_i \cdot \mathbf{C}'_{i.val}$ and increase the parent by $h_{\mathbf{C}'_{i.parent}} += h_i$. The RMM and LMM asymptotic runtime scale according to $\mathcal{O}(|\mathbf{D}| + |\mathbf{C}'|)$, making it very fast, when the number of unique *row-offset + value* pairs is low.

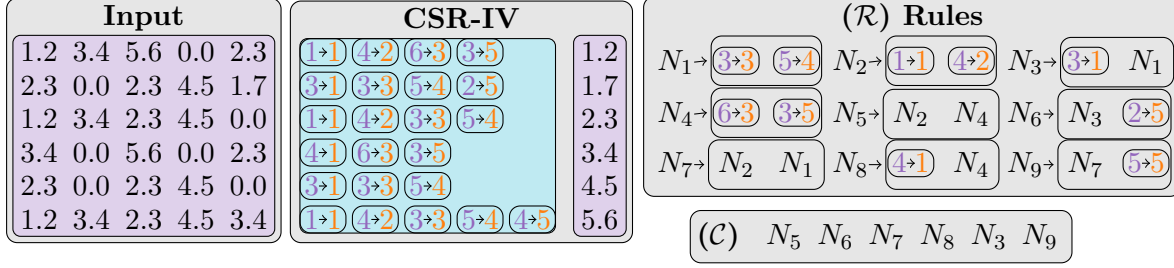


Figure 2.16: Grammar-based Compression Example (modified from [77]).

Table 2.4: Linear Algebra Asymptotic Runtimes of Aggregate and Binary Operations in GLA [77]. Missing operations compared to previous tables, are not supported.

Algebra	$\mathbf{A} + \mathbf{x}$	$\mathbf{A} \cdot \mathbf{c}$	$\mathbf{A} @ \mathbf{x}$	$\mathbf{x} @ \mathbf{A}$
GLA	\mathcal{DC}	$\mathcal{O}(d)$	$\mathcal{O}((\mathcal{R} + \mathcal{C})k)$	$\mathcal{O}((\mathcal{R} + \mathcal{C})k)$

2.4.3 Grammar-compressed Linear Algebra

Another related work is 'Improving Matrix-vector Multiplication via Lossless Grammar-Compressed matrices' [77]. This work introduces two compression techniques. The first, called CSRV, is equivalent to CSR-IV [130]. However, the second technique is new, using a grammar-compressed representation for linear algebra, henceforth abbreviated to *GLA*.

Compressed Format: Starting from the CSR-IV representation, the algorithm finds a grammar that can encode a CSR-IV matrix. Figure 2.16 shows a GLA compressed format. The compressed format consists of three parts. First, a set of base rules, also called the grammar, \mathcal{R} . Second, a sequence of rules, \mathcal{C} , the length of rows that decompose into individual rows. Third, the dictionary of unique values \mathcal{V} of size d .

Rules: A base rule is always a non-terminal pointing to two components. A component can either be a base-tuple or another rule. A non-terminal rule $N_i \rightarrow \mathcal{R}_{i_1} \mathcal{R}_{i_2}$ decomposes into a tuple of two other components. A base-tuple is equivalent to a tuple in CSR-IV, $\mathcal{T}_{\langle h, k \rangle}$, where h refers to a value index in \mathcal{V} and k a row-offset. Furthermore, the rules must be sorted with no rule points forward in the list of rules, such that N_i can point to N_j and N_k only if $j, k < i$. The sorting is essential for the specialized RMM and LMM in GLA.

Grammar-finding: GLA does not address the grammar-finding problem and instead uses an existing algorithm called RePair [138]. Finding a good grammar is an NP-complete problem [77, 49]. Grammar finding is also called a straight-line program (SLP) [155, 77]. Finding the smallest SLP is grammar-based compression [49, 159].

Runtime: Similar to TOC and CSV-VI, GLA relies on sparse linear algebra as a foundation. The worst-case performance of most operations depends on the number of base tuples in CSR-IV. However, if there are some repeated value row-offset pairs, GLA improves performance. The best-case performance is \log_2 of the number of tuples in rows because the branching factor of each rule in GLA is two.

Operations: GLA allows efficient sparse-safe scalar operations while decompressing its data in sparse unsafe operations, just like TOC and CSV-VI. An interesting case for GLA is matrix multiplication.

RMM: GLA use a process very similar to TOC. GLA simplify the RMM to process a column vector at a time. GLA allocate a temporary array, w , in the size of rules. For each i in 0 to $|\mathcal{R}|$ calculate N_i and add to w_i based on the two components of N_i . If a component is a base-tuple, calculate its output, $w_i += \mathcal{V}[\mathcal{T}_{key}] \cdot \mathcal{V}[\mathcal{T}_{col}]$. If the rule is a non-terminal, N_k , look up the already computed value of the rule ($k < i$) and add it $w_i += w_k$. Afterwards, to assign the output values, iterate through \mathcal{C} to compute the output vector by lookups into w . Overall each right-side column can be processed in $\mathcal{O}(|\mathcal{R}| + |\mathcal{C}|)$. Iterating through the rules works for the same reason as TOC.

LMM: LLM has a similar runtime to RMM and can employ the same trick as TOC processing, a single left-side row vector, y , at a time. First, allocate a w vector in the size of rules and an output vector \mathbf{R} in a number of columns of the GLA compressed matrix. Then, iterate through the \mathcal{C} and add the y_i values to the \mathcal{C}_i 's N rule index in w . Similar to TOC, iterate *backwards* through indices of \mathcal{R} . For each component q . If q is a non-terminal, N_j add w_i value to w_j (guaranteed $j < i$). Otherwise if q is a terminal $\mathcal{T}_{\langle h,k \rangle}$ increase \mathbf{R}_k by $\mathcal{V}_h \cdot w_i$.

Comparison to TOC: There are many similarities between GLA and TOC processing of matrix multiplications. Both employ ideas to push the computations through linked rules. GLA can possibly find a small set of rules that gives better performance than TOC. However, GLA has to perform an analysis and generate a grammar of rules, which can be costly and GLA's branching factor is two. In comparison, TOC's branching factor is unlimited and is able to dynamically rediscover its compression rules by leveraging the LZW compression technique. Both TOC and GLA optimize for the compression of rows.

2.4.4 Huffman Address Map

Huffman Address Map (HAM) and its sparsity exploiting version sparse HAM (sHAM) [157] also target efficient matrix-vector multiplication, specifically in the domain of deep neural networks. Unlike traditional sparsity exploiting techniques, sHAM considers sparsity a subclass of low-entropic data. The technique compresses the DNN weight matrices and relies on the matrices preprocessed with weight pruning and quantization [103]. As implied by the name, Ham is based on Huffman coding [111, 163].

LMM: The left matrix multiplication in HAM processes each row on the left independently via matrix-vector operations. For each column in the HAM compressed matrix, read through the HAM compressed bits, get the next entry's row index, look up its dictionary value, and multiply with the corresponding column value on the left. The runtime of HAM's vector LMM is $\mathcal{O}(nm)$, and sHAM is $\mathcal{O}(\mathbf{A}_{\neq 0})$. Each operation has a slight overhead compared to CSR, where the index lookup is replaced by a decoding of the Huffman codes, making the technique generally slower than a CSR equivalent implementation. However, HAM does improve memory usage and reduces power consumption in many use cases. Because the matrix-vector multiplication is memory bandwidth bound, it is possible to make a HAM implementation faster than uncompressed.

RMM: The HAM paper does not implement RMM but says it is similar to LMM.

2.4.5 Compressed Shared Elements Row

Another related technique for compressed operations is Compressed Shared Elements Row (CSER) [241]. The paper introduces two compression versions. First, the Compressed Entropy Row (CER) contains four arrays: First, a unique values array like in DDC, but sorted based on frequency. Second, a column indices array of column offsets like CSR. However, excluding the most frequent element and containing sequences for each unique value like in OLE. Third, a pointer array for the next element. Fourth and final, similar to CSR, an offsets list for the next new row starts. The second compression technique, Compressed Shared Elements Row (CSER), additionally stores a mapping (like DDC) to allow the unique values in arbitrary orders shared between compressions.

Performance: CSER's operational time complexity is equal to OLE, enabling the implementation to scale in the number of non-zeros because the compressed format is close to equivalent to OLE. However, unlike OLE in CLA [74], CSER does not have any cache-conscious blocking scheme.

2.5 Workload Adaptation

Most compression schemes optimize for size and/or (de)compression speed [90, 76, 51, 262, 239, 244, 55, 126], and adaptively chose various compression strategies based on the characteristics of the input data. Selecting encoding schemes depends on data, workload and underlying hardware [48]. The optimization goal makes sense for classical compression techniques, however when we can run operations directly on the compressed data we can optimize the data layout to efficiently process a workload.

2.5.1 Cost Modelling

Many in-memory database systems already use lightweight compression algorithms optimized for fast compression and decompression to support operations performance [4, 56, 126]. Damme et al. [59] conducted an extensive survey on cost-based selection strategies for integer-based compression techniques to evaluate different compression techniques performance for database systems. The survey highlights two measures to include in a cost-based selection: performance (decomposed into compression, decompression and processing speed) and compression rate (the relative size of the compressed data). The study concludes that there is “no single-best lightweight integer compression algorithm”, therefore, it further motivates for cost-based optimization and selection of compression techniques. Damme et al. [59] suggest selecting a compression algorithm \mathcal{A} , objective \mathcal{O} , and dataset \mathcal{D} . \mathcal{A} can be selected from a set of compression algorithms \mathbf{A} . \mathcal{O} is decomposed into t_{comp} compression time, t_{decomp} decompression time, t_{agg} aggregate or operation time and $rate_{comp}$ compression ratio. Selecting the optimal compression technique, \mathcal{A}_{opt} , then becomes a minimization problem of the cost of combinations of \mathcal{A} , \mathcal{O} , and \mathcal{D} [59]:

$$\mathcal{A}_{opt}(\mathbf{A}, \mathcal{O}, \mathcal{D}) = \underset{\mathcal{A} \in \mathbf{A}}{\operatorname{argmin}} \operatorname{truecost}(\mathcal{A}, \mathcal{O}, \mathcal{D})$$

The intention of the truecost function is to return the actual cost of an combination of \mathcal{A} , \mathcal{O} , and \mathcal{D} . To find the optimal \mathcal{A} , we would have to run the workload using all \mathcal{A} on \mathcal{D} . However, it is prohibitively expensive [134, 59] to evaluate all combinations of compression algorithms. Therefore, most approaches instead approximate the performance of \mathcal{A} based on \mathcal{O} and \mathcal{D} before selecting a specific technique. Damme et al. highlights two approaches to reduce the analysis cost: (1) rule-based, and (2) cost-based.

Rule-based: Rule-based selection strategies can be modeled as a decision tree [59], an example of rule-based selection is the static heuristics in Parquet [52, 134]. The rules can be based on properties of \mathbf{A} , \mathcal{O} and \mathcal{D} , where each node in the tree refines the selection of compression techniques. The downside of rule-based approaches, is that it does not necessarily select the optimal \mathcal{A} because some early rules might “rule out” better approaches too early.

Cost-based: Cost-based techniques try to estimate the cost of all \mathbf{A} independently, via comparable metrics of the cost of each technique. The typical comparison point is size, $rate_{comp}$, however some systems chose to choose \mathcal{A} based on any of the other properties [134, 90, 76] or compound weighted properties. If the cost-based approaches have perfect information of the entire input \mathcal{D} , it should be possible to select the optimal \mathcal{A} . However, similarly to applying all \mathbf{A} to \mathcal{D} collecting perfect information is also prohibitively expensive.

Our Approach: This thesis’s workload-aware compression planning uses a bit of rule-based selection to introduce compression instructions into user defined scripts and cost-based compression based on summaries of linear algebra programs in order to tune *online* lossless matrix compression and compressed operations.

2.5.2 Property Estimation

For both rule- and cost-based approaches we want to estimate the performance of \mathcal{A} before applying it to data. A common approach is to extrapolate the performance of \mathcal{A} based on properties from the data [59, 134, 74]. Min, max, and unique value frequencies are enough for simple compression techniques such as FOR, DDC and constant compressions [134]. However, other techniques, such as RLE, require different information, such as the number of runs of similar values. Therefore, the set of properties needed is dependent on the compression techniques selected.

Maintained Statistics: Some systems, data structures, and files maintain properties of the entire datasets such as non-zero values or min-max ranges, making cost-based \mathcal{A} selection simple if \mathcal{A} does not require further information. While SystemDS does maintain the number of non-zero values, it unfortunately does not help the CLA implementation much because CLA needs more fine-grained information such as non-zeros per column that could be available if the system would use compressed sparse columns (CSC).

Cardinality Estimation: One of the most important components to estimate is the number of distinct values d . Cardinality estimation is notoriously challenging since exact counts using a can set cost memory equal to the input size if all values are unique. Approximate methods, such as HyperLogLog, calculate an approximate distinct value count with fixed memory budgets [104]. However, full dataset iterations can be prohibitively expensive for repeated analysis of multiple encoding plans [74].

Sample Estimation: To efficiently approximate the data's properties an approach is to extract a sample of the data. A set of informative properties that can be collected from a sample, and extrapolated to statistics for the entire dataset [100]. However, sample estimation approaches are not guaranteed to find the correct statistics. It can easily be proven that the estimations can be arbitrarily bad if the remaining data not sampled is completely different [100].

Sample-based Cardinality Estimation: When counting the number of distinct values in a sample we reduce the upper bound of memory, but introduce uncertainty. The distinct count has an obvious lower bound based on the distinct values in the sample and the natural upper bound of the total number of values in the input. CLA use Haas et al. [100] an extension to a generalized jackknife approach [96] that estimates d in a finite-sized collection.

Sample Selection: The sample can be selected in many ways, with different tradeoffs. If the sample takes the first k rows of the dataset, it is very efficient because it does not have to scan or jump through the dataset. However, taking the first elements can be very biased [134]. An unbiased sample is a sample without replacement, that with uniform randomness chooses elements without duplicate indexes. However, while random samples work for estimating many properties, it does not preserve the locality of the data [134]. Other sample-based approaches can improve the selection to improve run counting for RLE, such as extracting small continuous runs as samples [134].

Learned Selection: It is also possible to skip handcrafted properties for selecting compression schemes and instead rely on a learned feature vector, an example is Learned Encoding Advisor (LEA) [48]. Based on a sample of data, and simple statistics LEA can select the encoding type optimizing for size or query performance. LEA is trained to predict the encoded cost including encoded size, memory speed and storage speed for processing different \mathcal{A} , given a sample and a few collected statistics.

Constrained Selection: Another important aspect in selecting compression schemes is to adhere to constraints while optimizing for other goals. A good example is selecting \mathcal{A} to make the data fit in memory, but once it fits optimize for performance. This is for instance addressed in "Workload-Driven and Robust Selection of Compression Schemes for Column Stores" [38], that additionally considers the robustness to workload-shifts of the compression configuration in the context of the Hyrise system [70].

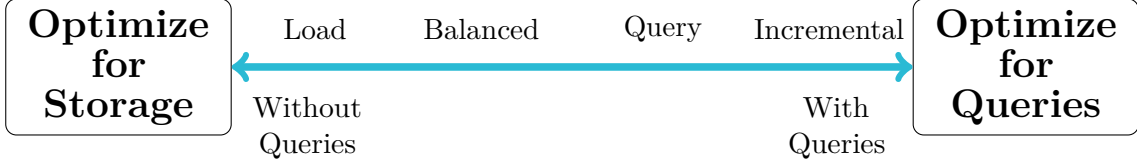


Figure 2.17: Storage vs Query Tradeoff [230]

Our Approach: This thesis’s workload-aware compression uses samples to collect statistics to estimate the performance of linear algebra operations and in turn the compressed format for entire linear algebra workloads.

2.5.3 Multi Objective Optimization

Figure 2.17 (Vertica [230]) highlight the tradeoff space of their DBDesigner, a customizable physical design tool for the Vertica analytic database. While the focus in that paper is on running a database while optimizing either towards query performance or storage space, it also is valid for compressed linear algebra operations.

Storage Optimization: When the workload adaptation should optimize for storage space—via, for instance, more heavyweight compression techniques—it would cost more to decompress the data, and therefore queries would be slower. In the most extreme case, there would be no query workload to be performed on the data, e.g. logging data that never is used or even loaded again.

Query Optimization: When moving from only stored data, the second level is to load the data, while performing little to no queries. Here it still makes sense to save data heavily compressed because of its rare usage. Moving over to the other side, if the data is frequently modified with incremental updates it is hard to keep a unified compression scheme and the data therefore would not benefit from compression. In such cases other data formats that optimize for the workload of frequent queries is better [137].

Storage Pruning: All queries do not need to process all data in DBMS systems, therefore DBMS systems can strategically load only effected blocks of data from storage. A recent example of this is in learned partitioning schemes that aim to maximize partitioning pruning [253]. However, unlike DBMS systems this thesis loads entire datasets, because we rely on full operations on all data.

Our approach: This thesis’s workload-aware compression optimizes towards both directions. When we save compressed data to disk, we optimize for storage space and I/O time, leveraging lightweight compression techniques. We optimize for storage and I/O because when data is written, it is unknown what future program workloads look like. Once we read data, we modify the data into a compute-oriented compressed format that adapts based on a user-defined linear algebra program.

2.6 Feature Engineering

Data-centric ML pipelines extend ML pipelines with additional preprocessing steps. These extending techniques can substantially improve model accuracy [131, 212, 248, 67], generalizability [7], and other measures like fairness [203, 219]. In this section, we highlight techniques and influential works.

Definitions: We differentiate between arithmetic transformations, which we can efficiently map to linear algebra operations on entire input matrices and feature engineering, which we cannot. However, common for most transformations, there is a “*build*” process that collects information about input data to then “*apply*” the transformations equivalently on unseen data. In SystemDS, we call these **transformencode** to build and apply the transformations, and **transformapply** to only apply the transformations.

Scope of Interest: To not cover the whole field of feature engineering, we focus on the transformations from a point of sparsifying data or introducing redundancy we can exploit in compression. As shown in the motivation (Section 1.2) and in the related compression techniques (Section 2.1), these properties can improve computational efficiency. However, some preprocessing operations densify the input, making sparsity exploitation impractical.

Automation: There is a high degree of complexity in choosing the right transformations and augmentations to perform on data to improve model accuracy. Therefore, many systems propose automated iterating and selecting the preprocessing pipelines [203, 256, 8, 212]. AutoML tools such as Auto-WEKA [225, 129] and Auto SKlearn [79, 78] include easy-to-use abstractions for end users.

2.6.1 Categorical Transformations

Dictionary: Some feature transformations build dictionaries that map unique input values to new values, called dictionary encoding. This transformation is typically done on categorical values to transform them into numeric values. However, it can also be applied to numerical inputs. An example is recoding values, where each unique value in an input is recoded to contiguous integers. Categorical data can be further subdivided into nominal data, and ordinal data. Nominal data is unordered, while ordinal data can be ordered. Nominal data transformed via dictionary encoding is typically combined with a one-hot encoding transformation, while nominal data can return its numeric ordered values. Dictionary encoding does not introduce zero values and, therefore, does not increase the sparsity. Similarly, dictionary encoding does not reduce the number of distinct values since each unique input value is mapped to a single new value. However, when combined with one-hot encoding, the sparsity does increase linearly with the number of unique values $1/d$, as shown in the motivation Section 1.2.3.

Hashing: Instead of building a dictionary of all unique values, a hash based encoder can be used. As mention in Section 1.2.3, feature hashing maps values to Δ buckets, making the upper bound of unique values after encoding controllable. There are efficiency benefits to feature hashing compared to dictionary-based approaches. The most notable is no dictionary allocation. Furthermore, hash encoding can compute each value independently, meaning the build phase of transform encode is free. A downside is that hashing is a lossy transformation. Hash collisions are likely with small Δ compared to distinct inputs due to the birthday problem (assuming a uniformly distributing hash function). From the point of view of sparsity exploitation hash encoding have the same properties as dictionary encoding. However, with one difference being the sparsity, and in turn, the number of output columns, is controllable, $1/\Delta$, when combined with one-hot encoding.

2.6.2 Arithmetic Transformations

Many transformations modify the numeric values of data in data-centric pipelines. Examples of such transformation are min-max normalization, standard scoring (also called z-score), and winsorization (or clipping). Clustering algorithms can also be used to find other features, such as K-means or PCA. Transformations of images and sound, such as affine transformations or Fourier transformations, are also part of arithmetic feature transformations. All these operations can be performed via standard linear algebra operations, such as binary scalar operations, matrix multiplies and min-max-sum aggregates.

Normalization: Before fitting a model to data, most pipelines include a normalization step. The normalization is typically done globally or on individual features. The global normalization aggregates all values, while individual features typically apply column aggregates. However, many of these invalidate sparse linear algebra because they densify the input data. Min-max normalization scales all values to fall in the range of 0 and 1. If the minimum value is zero, then the sparsity is maintained, or if the minimum value is overrepresented in the data, then the returned data can be sparsified. The standard score, is calculated via $\hat{\mathbf{X}} = (\mathbf{X} - \mu)/\sigma$, where

μ is the mean and σ is the standard deviation equal to $\sigma = \sqrt{(X - \mu)^2 / |X|}$. SystemDS's scale function performs z-score with Bessel's correction on individual features, making the operation $\sigma_i = \sqrt{(\mathbf{X}_{:,i} - \mu_i)^2 / (|\mathbf{X}_{:,i}| - 1)}$. Similar to min-max normalization, z-score normalization densifies and only maintains its sparsity if the mean of input features is zero.

Clipping: Clipping or winsorizing input features tries to remove outliers data. Clipping replace all values above or below a specific value $\hat{\mathbf{X}} = \min(\mathbf{X}, c)$ or $\hat{\mathbf{X}} = \max(\mathbf{X}, c)$ with the boundary value. While winsorizing sets the clipping bounds based on quantile ranges, typically the 5% and 95% quantiles. Both types of clipping maintain or improve sparsity while also maintaining and improving the number of distinct values, making clipping work nicely with sparsity exploiting techniques as well as CLA.

Binning: Section 1.2.2 introduced and motivated for learned [258, 257] and static quantization schemes/binning techniques. Binning maintains or improves sparsity and fully controls the number of distinct values.

Clustering: Using clustering algorithms allows us to control the overall dimensionality of the input. PCA does clustering by selecting only the top-k most influential principal components. However, PCA, unfortunately, tends to densify and increase the number of unique values. Therefore, PCA does not work well as a preprocessing step with compression. We introduce ways to make PCA return compressed formats. K-means can control its output dimensions by setting the number of centroids used. We can use either the closest centroid as a categorical feature or the distances to all centroids as a feature for other ML models. In the first case, it is fully controllable how many distinct values we get, while the second tends to return dense outputs not amenable to compression.

Image Affine Transformations: Affine transformations of images modify the location of the pixel values via a transformation matrix. When a matrix multiplying the transformation matrix with an input coordinate, an output coordinate can be determined. With this in mind, it is possible to define a sparse 'selection' matrix filled with coordinate mappings that, when multiplied with the input image, apply the affine transformation. Depending on the interpolation type applied, affine transformations can maintain sparsity and the number of distinct values. However, commonly more complex interpolation functions improve the output quality from affine transformations but do not maintain the distinct values.

2.6.3 Feature Augmentations

While transforming existing features is useful, ML models can leverage multiple features with different preprocessing steps by collecting these features together into a larger dimensional input space. Additionally, modifying the input features allows simple models, such as linear models, to find solutions because of additional non-linear appended or modified features.

Non-Linear Example: Given a list of vectors $\mathbf{X} \in \mathbb{R}^{n \times m}$ and $y \in \{c_1, c_2\}$, we can train a linear model to distinguish two classes in \mathbf{X} via $D(x_i) = \mathbf{W}x_i + w_0$. If $D(x_i) > 0$ then $x_i \in c_1$ otherwise $x_i \in c_2$ [42]. Many solutions exist, and therefore algorithms, for finding weights \mathbf{W} and intercept values w_0 that try to split the classes. Unfortunately, linear models are not always a good fit because no hyperplane can split the two classes accurately. We can apply non-linearities to the input data before fitting the mapping to improve the model. For instance, it might be that $D(x_i) = \mathbf{W} \log_2(x_i) + w_0$ is a better fit.

Kernels: Kernel machines, where SVM is the most known member [42], formalize the technique of extending linear classifiers and solve non-linear problems via kernel functions [15]. A kernel function measures the similarity between two points x_i and x_j defined as $\mathcal{K}(x_i, x_j) = \varphi(x_i)^\top \varphi(x_j)$. $\varphi : \mathbb{R}^m \rightarrow \mathbb{R}^q$ is a function that maps the input space of dimension m to the feature space of dimension q . A simple example is $\varphi_a(x_i) = \varphi_a([x_{i1}, x_{i2}]) = [x_{i1}, x_{i2}, x_{i1}^2 + x_{i2}^2]$ where $x_i \in \mathbb{R}^2$ expands to $\varphi_a(x_i) \in \mathbb{R}^3$. The kernel function for φ_a in such a case simplifies to $\mathcal{K}_a(x_i, x_j) = x_i \cdot x_j + \|x_i\|^2 \|x_j\|^2$ (proof in Appendix A.4).

Primal and Dual: When a kernel function is available, it is possible to find the weights for W and w_0 with a primal and dual solution [42]. We want to find the weights:

$$D(x_i) = \mathbf{W}\varphi(x_i) + w_0$$

A primal space solution requires materializing the result of applying φ to all inputs in \mathbf{X} and then using some algorithm to solve for \mathbf{W} and w_0 . However, the dimensionality of $\varphi(x)$ can be very large because the feature space of \mathbf{X} grows based on the applied non-linearities. Alternatively, we can solve the equations indirectly in dual space by finding another weight matrix \mathbf{Z} via applying the kernel function \mathcal{K} on pairs of input tuples. Solving in dual space can scale much better than using φ .

$$D(x) = \sum_{k=1}^n \mathbf{Z}_k \mathcal{K}(x_k, x) + w_0$$

Under some conditions [42], it is possible to rediscover \mathbf{W} from \mathbf{Z} via the dual representation:

$$W_i = \sum_{k=1}^n Z_k \varphi(X_k)$$

Polynomial Expansion: A common non-linearity is a polynomial expansion $K(x, x') = (x \cdot x' + c)^p$ of order p . If the order is 2 the φ mapping expands to:

$$\varphi_p(x) = (x_n^2, \dots, x_1^2, \sqrt{2}x_n x_{n-1}, \dots, \sqrt{2}x_n x_1, \\ \sqrt{2}x_{n-1}x_{n-2}, \dots, \sqrt{2}x_{n-1}x_1, \dots, \sqrt{2}x_2x_1, \sqrt{2}cx_n, \dots, \sqrt{2}cx_1, c)$$

As can be seen in the equation, the number of additional features in $\varphi(x)$ grows quickly. There are $((n(n+1))/2) + n$ parameters, and when increasing p the scaling of $\varphi_p(x)$ is $\mathcal{O}(n^p)$. However, it is possible to skip allocating and computing this larger space by using the dual space and the simplified formula from the kernel function. Using the dual space while avoiding allocation of the kernel space is known as the kernel trick while solving the linear models in the transformed space is known as quadratic programming [246].

Big Feature Space: The feature space of φ can, in theory, be infinite with the right kernel function. Additionally, kernel machines and a polynomial fitted to the same degree as instances in \mathbf{X} are guaranteed to be able to perfectly map to \mathbf{Y} [15]. These two properties together make computing using the dual space tempting, even with the squared allocation of the kernel function.

2.7 Summary

This chapter introduced the conceptual foundations and related work for the contributions proposed in the subsequent chapters. We covered individual lightweight and heavyweight compression techniques, system integration and hybrid approaches. We show how related work pushes operations into custom monolithic compression schemes, column-based compression techniques, and fine-grained sparsity exploiting structures. We identified a common weakness of related compressed linear algebra work relying on sparse linear algebra. Related work, therefore, suffers from dense inputs or densifying operations on their sparse compression techniques. We highlight workload analysis techniques and adaptations to optimize compressed formats while identifying a new optimization potential for compressed operations. We covered different feature transformations and engineering while focusing on their effects on sparsity and distinct values. These transformation's effects are ideal matches for compressed operations.

3

Workload-aware Compressed Linear Algebra

Parts of this chapter are published in AWARE [23] and under submission in BWARE.

This chapter introduces our contributions to performing linear algebra operations on compressed data. Previous work on compressed linear algebra (CLA) [74, 75] allowed for general redundancy-exploitation (with repeated values and correlation) by applying lightweight lossless compression techniques like dictionary, run-length, and offset-list encoding and executing linear algebra operations like matrix-vector multiplications and element-wise operations directly on compressed representations. CLA was integrated into Apache SystemML [33], but by default, only applied for multi-column matrices, whose size exceeded aggregated cluster memory, and only a small set of operations (which were commonly executed on the large compressed feature matrix) were supported in compressed space. The constraint of exceeding aggregate cluster memory ensures that online compression time and space overheads are amortized but limit applicability in practice.

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

- *Compression Framework:* New encodings and compressed operations (Section 3.1 and Section 3.2), designed for compressed intermediates and thus, chains of operations.
- *Workload-aware Compression:* Novel workload-aware compression planning and compilation techniques (Section 3.3).
- *Experiments:* Local and distributed experiments comparing uncompressed linear algebra (ULA), CLA [74, 75], TensorFlow, and AWARE on various workloads (Section 3.4).

Our workload-aware compression addresses the limitations from CLA with new techniques for compression planning, compressed intermediates, and different optimization objectives and all contributions are fully integrated into Apache SystemDS [31] and deemed reproducible¹. Table 3.1 highlights some of the key differences between CLA and AWARE.

¹<https://github.com/damslab/reproducibility/tree/master/sigmod2023-AWARE-p5>

Table 3.1: Key Differences of CLA and AWARE

	CLA [74, 75]	AWARE
Co-Coding	$\mathcal{O}(m^2)$	$\mathcal{O}(m)$
Column Group Encodings	4 (5)	7 (327)
Materialization	Eager	Deferred
Optimization Objective	Data	Data & Ops
Matrix Multiplication	MV, VM	MV, VM, MM

- First, to reduce compression time, we introduce a new co-coding technique that performs group *combinations* instead of group *extractions*, reducing the overhead of analyzing groups. Our co-coding approach includes a new enumeration heuristic that only evaluates $\mathcal{O}(m)$ group combinations.
- Second, for extended utilization of compressed intermediates, we provide new column group encodings that facilitate shallow-copy operations. Table 3.1 shows the number of high-level column group types and—in parenthesis—the total number of variants of these encodings. We also introduce a deferred operation/encoding design, where compressed operations can output different types of encodings, allowing compressed intermediates where CLA would decompress or return inefficient representations. Furthermore, AWARE natively supports compressed matrix-matrix multiplication (even with two compressed inputs), unlike CLA, which would process it via repeated matrix-vector multiplications and thus decompresses one side.
- Finally, above all, AWARE uses a cost-based optimization objective of minimizing the workload execution time and tuning the compression process, the compressed representation, and compressed operations in a principled way.

Notation: For this chapter, we need some additional specific notation. An $n \times m$ uncompressed input matrix is compressed into a set of column groups \mathcal{G} , where $|\mathcal{G}|$ denotes the number of column groups (with $|\mathcal{G}| \leq m$ without overlapping groups), and \mathcal{G}_i denotes the i -th column group. A single column group \mathcal{G}_i comprises \mathcal{G}_{ic} columns, a $d_i \times \mathcal{G}_{ic}$ dictionary \mathcal{D}_i with d_i distinct tuples, and an index structure \mathcal{I}_i . $\sum_{i=1}^{|\mathcal{G}|} \mathcal{G}_{ic} \geq m$ must be true for the validity of operations in AWARE. Furthermore, \mathcal{F}_i is the frequency of each tuple in \mathcal{D}_i based on \mathcal{I}_i . For matrix multiplications $\mathbf{A}\mathcal{G}$ or $\mathcal{G}\mathbf{B}$, let k denote the number of rows in \mathbf{A} and columns in \mathbf{B} , respectively. Given a matrix or vector \mathbf{X} , $\mathbf{X}_{\neq 0}$ is the set of non-zero values in the matrix, while $(\mathcal{G}_i)_{\neq 0}$ is the non-zeros in an uncompressed version of the column group. Some encodings use default tuples, for which we repurpose μ . We write $\mathcal{G}_{i\mu}$ as the set of rows containing the default μ in an uncompressed version of the i -th group, and the compliment $\mathcal{G}_{i\mu}^c$ is the set of rows not containing the default tuple μ . If the encoding does not use default tuples then the compliment $|\mathcal{G}_{i\mu}^c| = n$ is equal to the number of rows (which is true for DDC) and $|\mathcal{G}_{i\mu}| = 0$.

Table 3.2: Overview of CLA and AWARE Column Groups.

Type	Description	CLA [74, 75]	AWARE
CON	Constant or Empty Columns		✓
DDC	Dense Dictionary Coding	✓	✓
OLE	Offset-list Encoding	✓	(✓)
FOR	Frame of Reference		✓
RLE	Run-length Encoding	✓	(✓)
SDC	Sparse Dictionary Coding		✓
UC	Uncompressed (dense/sparse)	✓	✓

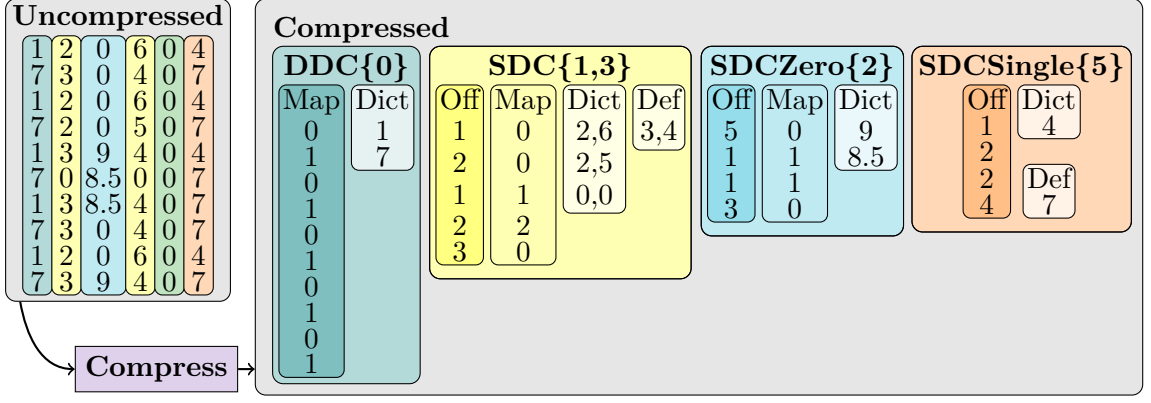


Figure 3.1: Example of AWARE Matrix Compression.

3.1 Compression

This section describes AWARE’s compressed representation, selected new concepts, and the overall compression algorithm. The new encoding schemes are designed for redundancy-exploitation across operations, while the new compression algorithm ensures fast, easy to amortize compression. AWARE encodes each column group independently in a specific encoding type. Table 3.2 shows these column-group encodings, as well as the differences to CLA. Figure 3.1 then presents an example of compressing a 10×6 matrix into three single-column groups (0, 2 and 5), one two-column group ($\{1, 3\}$), and an empty group. The concept of column groups is exactly the same as in CLA.

3.1.1 Dictionary-based Compression

Most of the encodings are already introduced in Section 2.1, and this chapter only introduces the new encoding types and highlights the differences in encoding specific for AWARE. Notably, AWARE encode column groups of a matrix in different encodings, such that we can specialize the compression depending on column properties.

Dense Dictionary Coding (DDC): A DDC column encoding contains two parts: a *dictionary* with the distinct value tuples in the column group, (shown in Figure 3.1 as a Dict with 2 values for column $\{0\}$), and an *index structure* with a row-to-tuple mapping (e.g., dictionary position). DDC is dense because each row input is assigned a code in the map.

Sparse Dictionary Coding (SDC): SDC is a new encoding, it is a combination of DDC and sparse matrix formats like compressed sparse rows (CSR). An example is shown in yellow for columns $\{1, 3\}$ in Figure 3.1. Like DDC, each group has a dictionary of all unique tuples except the most frequent tuple named “Def” for default. This scheme encodes row locations of non-default tuples in the index structure as row-index pairs. This approach is similar to compressed sparse columns (CSC) that store row-index/value pairs for non-zero values, but extends it for general redundancy-exploitation (default values, dictionary references). In many ways it is also similar to many of the other sparsity-exploiting compression schemes such as CSR-IV, however, with major performance differences. The first difference is that the row part is further specialized to delta offsets (“Off”) from previous rows to allow smaller physical codewords. The second difference is the maintaining of a default value. Similar to CSR-IV, a “Map” maps offsets to tuples in the dictionary. If we define z_i as the number of tuples in \mathcal{G}_i with at least one non-zero value $z_i = ((\mathcal{G}_i), \exists x \neq 0)$ (similar to OLE in Section 2.1). Then, SDC scales according to $\mathcal{O}(d\mathcal{G}_{ic} + z)$, which is equivalent to OLE without the blocking scheme.

SDC Specializations: SDC specializes into SDCZ, Z for Zero, where the default tuples only contain zero values, meaning sparse linear algebra rules apply. SDCZ is shown in the $\{2\}$ blue column group. Another specialization is SDCS, S for Single, binary data (one dictionary entry, one default), In the SDCS case, there is no need for mapping codes because all offsets map to the non-default value. An example of SDCS is the $\{5\}$ orange column group.

Frame of Reference (FOR): We use FOR as a second layer on top of DDC or SDC (called DDCFOR, SDCFOR). This encoding shallow-copies the index structures and dictionaries, and allocates a reference tuple, that indicates a global value offset. An SDC group can zero-out the default tuple by adding it to the dictionary and subtracting it from the reference tuple and converted to SDCZ, meaning dense compressed formats still can exploit the common elements and perform similarly to sparse linear algebra.

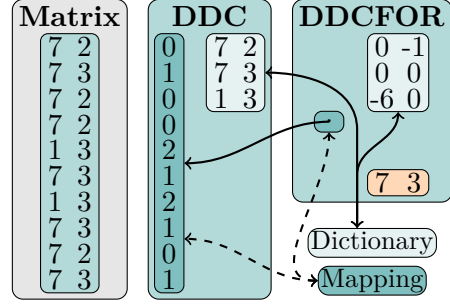


Figure 3.2: DDCFOR scheme

Constant Encodings (CON): CON encodings are used for empty, constant columns, and constant tuple column groups. CLA encodes such groups using run-length encoding (RLE). However, RLE is inefficient because CLA would require runs to be processed, while most linear algebra operations would perform much more efficiently on constant groups. Instead, we specialize with constant groups in order to simplify operations with compressed outputs, and leverage the constant groups in some operations to set constant global offsets and inject sparse operations to otherwise dense column groups.

3.1.2 Index Structures for Compression

Dictionaries: CLA uses basic FP64 (double) dictionaries. In contrast, AWARE generalizes the data binding of dictionaries and uses basic FP64, sparse CSR matrices, and specialized identity matrices. The more columns co-coded, the more zeros might be included in unique tuples and thus, warrant a sparse dictionary. AWARE does not share dictionaries across multiple column groups like CLA does in some cases.

Index Encodings: The different column group implementations share common primitives such as Map and Off, of different value types (not shown in the figure). Map supports encodings in Bit, Byte, UByte, Char, Char+Byte and Int, while Off supports delta-encoded Byte, UByte or Char arrays, and specializations for one/two offsets. Once column groups encode thousands of columns together, the column indexes can be the bottleneck. Therefore, we also have specializations for column indexes in form of range indexes, and recursive indexes for joining column groups without extra allocations.

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

3.1.3 Online Compression Sequence

Our compression algorithm aims to reduce the online compression² time, introduce workload-awareness via generic cost functions (computation, memory or combinations), and handle matrices with many columns. Together, solutions to these issues, allow us to apply compression for a wide variety of inputs and intermediates with robust performance improvements. Given an uncompressed matrix, the AWARE compression algorithm as shown in Figure 3.3 and Algorithm 1. The figure highlight structural differences from CLA with red, however individual parts have also been reworked. The AWARE compression comprises the following phases:

a) Classifying: For efficient compression planning, we first obtain an index structure (dense or sparse for DDC or SDC) for each column in a sample of the input matrix, as well as counts of non zeros (NNZ) per column in the input matrix. Using the index structure and NNZ count, we compute summary statistics for individual columns (e.g., the frequency of

²Online compression refers to the compression of inputs or intermediates during runtime of a linear algebra program (e.g., after reading uncompressed inputs).

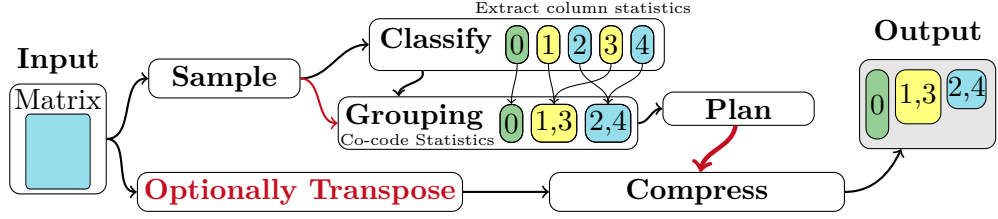


Figure 3.3: AWARE Compression Sequence, comparable to CLA Figure 2.11.

Algorithm 1 AWARE Compression Algorithm**Require:** Matrix input $\mathbf{X} \in \mathbb{R}^{n \times m}$ **return** \mathcal{G}

$\mathcal{I}^s \leftarrow \text{EXTRACTINDEXSTRUCTURES}(\text{SAMPLE}(\mathbf{X}))$ $\triangleright \mathcal{I}^s$ is the index sample, $|\mathcal{I}^s| = m$
 $\mathcal{G}^q \leftarrow \text{CLASSIFY}(\mathcal{I}^s)$ \triangleright Abort location, \mathcal{G}^q is the estimated compression, $|\mathcal{G}^q|$ is here $= m$
 $\mathcal{P} \leftarrow \text{GROUPING}(\mathcal{G}^q, \mathcal{I}^s)$ \triangleright Abort location, \mathcal{P} is the compression plan
 $(\mathbf{M}, t) \leftarrow \text{TRANSPOSE}(\mathcal{P}, \mathbf{X})$ \triangleright If t is true then $\mathbf{M} = \mathbf{X}^\top$ else $\mathbf{M} = \mathbf{X}$
 $\mathcal{G} \leftarrow \text{FINALIZE}(\text{COMPRESS}(\mathcal{P}, \mathbf{M}, t))$ \triangleright Final abort location

distinct items), estimate the cost of the individual columns, classify columns as compressible or incompressible, and extract empty columns. For classifying a column or list of columns, the same summary statistics are needed, irrespective of optimizing for workload cost or size in memory. Compared to the CLA compression algorithm—where the entire uncompressed matrix was transposed first for efficient extraction in Classify and Compress—we benefit from working only with small index structures until deciding on aborting the compression for non-amenable matrices. Furthermore, we gain more efficient sample extraction, and bounded temporary memory requirements for incompressible matrices.

b) Grouping: Column co-coding seeks to find column groups in order to exploit redundancy among correlated columns. AWARE introduces two techniques to improve CLA’s co-coding algorithm. First, instead of extracting statistics from the sample when combining columns, we combine the index structures of two already extracted groups from the classification phase or previously combined columns. Algorithm 2 combines two dense index structures (\mathcal{I}^r and \mathcal{I}^l) into a combined index structure \mathcal{I}^c . This algorithm allocates a mapping M that is able to encode all possible unique mappings from combining \mathcal{I}^r and \mathcal{I}^l by the product of their numbers of distinct items d_l and d_r . Further specializations are algorithms for sparse-sparse and sparse-dense combining. Second, we introduce a new co-coding algorithm (see Algorithm 3) that uses a priority queue Q for sorting columns (or column groups) based on a configurable cost function, and combines groups at the head of the queue. We found that starting with this new co-coding algorithm and switching to a greedy combining approach at a threshold number of remaining groups gives a good balance of compression time and quality. In cases with millions of columns, we do a static partitioning of the columns to available threads and combine columns in a thread-local manner.

Algorithm 2 Combine Algorithm for Dense Index Structures**Require:** Index structures for two groups $\mathcal{I}^l, \mathcal{I}^r$ **return** Combined index structure \mathcal{I}^c

$M \leftarrow \mathcal{I}[d_l \cdot d_r]$, $u \leftarrow 1$ \triangleright Allocate map of possible distinct size
for $i \leftarrow 0$ to n **do**
 $m \leftarrow \mathcal{I}_i^l + \mathcal{I}_i^r \cdot d_r$ \triangleright Calculate new unique index
if $M_m = 0$ **then** \triangleright Non-existing value at the unique index
 $M_m \leftarrow u++$ \triangleright Assign unique index to next unique value
end if
 $\mathcal{I}_i^c \leftarrow M_m - 1$ \triangleright Assign output to map value at unique index
end for

Algorithm 3 PriorityQueue Co-coding Algorithm

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

Asymptotic Change: Both index structure combining and the new co-coding algorithm modify the asymptotic execution time of the column grouping. Combining the index structures makes the worst-case dense operations $\mathcal{O}(n)$ to combine two column groups if n is the number of rows in the sample, rather than $\mathcal{O}(|n(\mathcal{G}_{i_c} + \mathcal{G}_{j_c})|)$ that adds the number of columns of either column group combined. If both sides use sparse index structures, then the runtime is better $\mathcal{O}((\mathcal{G}_i^c \mu) + \mathcal{G}_j^c \mu)$ since we only have to process uncommon elements of each. Similarly, the co-coding algorithm changes the runtime of combining columns from a greedy $\mathcal{O}(m^2)$ to $\mathcal{O}(m)$ because each iteration of the while loop reduces the queue size by one and the queue is guaranteed to start at m column groups.

c) Transposing: The uncompressed input matrix can be transposed (columns in row-major) if the compress phase would benefit from sequential access and amortize such data reorganization. This decision is dependent on the data characteristics (e.g., matrix dimensions, dense or sparse) and the chosen compression plan (e.g., co-coded columns). In general, it is more efficient to compress column groups with many columns in a non-transposed input, while for few column column groups, is better with transposition. At the time of writing the thesis the transposition rules is never to transpose if the input is a dense matrix, while if it is sparse only transpose if the number of columns and rows is large, if the number of non-zeros is low, or if there is more columns than one per 30 column groups. The decision to transpose can be further tuned depending on the selected encodings from the compression planning, but the decision tress heuristics is applied currently. The AWARE paper had slightly different rules [23], that have been updated based on more specializations of compressions in the following compression phase.

d) Compressing: During compression, we take the input matrix and compression plan (co-coding decisions, and column-group types), and create the compressed column groups. CLA would first extract its single- or multi-column uncompressed bitmap as a canonical representation of distinct tuples and offset lists per tuple. With these temporary offset lists, it would re-evaluate the group types, and finally create the physical encoding of the compressed column groups, which involves various specializations (e.g., delta-encoded offsets) for smaller code words. However, AWARE instead skips the bitmap construction in most cases, and instead directly allocates the requested compression type while having a fallback to CLA’s bitmap if it fails. An example is the directCompressDDC method, that first allocates a mapping that is guaranteed to be able to hold up to number-of-rows unique elements, and then (possibly in parallel row blocks) parse rows and collect unique tuples for a dictionary and assign the map in a single pass of the input matrix. Once all rows are processed, we allocate the dictionary, and resize the mapping to an optimized allocation. Once any column group is compressed—and it is beneficial in terms of workload costs—we analyze if we can sparsify its dictionary via a FOR encoding, and if so apply the transformation. In contrast to CLA, we apply no corrections for estimated compressible but actually incompressible columns because the estimators and co-coding show robust behavior.

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

Parallelization Strategies: When compressing distributed matrices, blocks are compressed independently in a data-parallel manner with single-threaded compression per block. In contrast, local, in-memory compression utilizes multi-threading with barriers per phase. Classify parallelizes over columns, Grouping over blocks of columns, Compress over column groups and in some cases row partitions, and Transpose uses a multi-threaded cache-conscious uncompressed transpose operation. A more fine-grained parallelization with a task graph [167] is interesting future work.

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

3.1.4 Compressed Design Principles

As a basis for discussing compressed operations, we first summarize underlying design principles.

Definitions and Scope: We define *sparse-safe* operations as operations or aggregations that only need to process non-zero input cells. For example, $\text{round}(\mathbf{X})$ is sparse-safe, while $\text{exp}(\mathbf{X})$ is sparse-unsafe because $\text{exp}(0) = 1$. *Special values* like NaN (not-a-number, with $\text{NaN} \cdot 0 = \text{NaN}$) are not supported in compressed operations because they render sparse linear algebra invalid [215]. We allow compressing matrices that contain NaN values and further allow users to replace all instances of NaN with numbers on the compressed matrices, but if the compressed matrices contains NaN the results are not guaranteed.

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

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

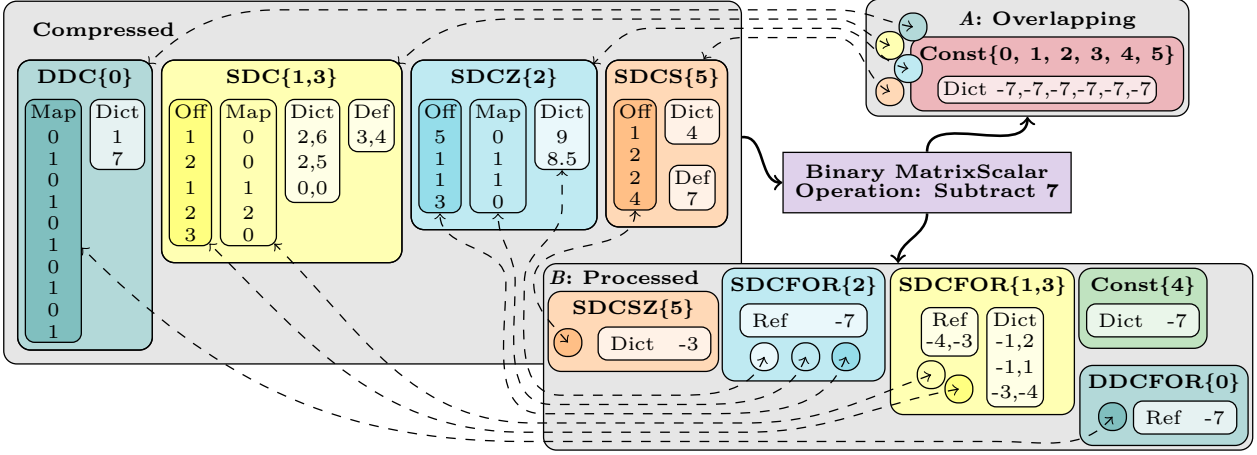


Figure 3.4: Example of AWARE Binary Subtract 7 Operation Matrix Compression.

3.2 Compressed Operations

The compressed AWARE operations are modifications of the original CLA operations. While an initial look to some operations might not seem novel, it is the set of changes of the entire set of operations and novel techniques for delaying computation as much as possible that really makes it fundamentally different from the design and approach in the original CLA papers and implementation.

3.2.1 A Motivating Operations Example

Figure 3.4 shows an example of performing a binary elementwise subtraction of 7 to all cells. The input is the compressed matrix from Figure 3.1. The figure shows two alternative compressed outputs.

Option A: Option A constructs an overlapping result. The overlapping result contains pointers to the input compressed block’s column groups, and a new constant column group is overlapped onto the compressed output. The new overlapping group contains the constant -7 that is subtracted from all columns. The overlapping state allows us to additively decompress all column groups on top of each other to produce the uncompressed matrix. This first option only allocates and assigns a vector in the number of columns of the compressed input, $\mathcal{O}(m)$.

Option B: Option B shows how pushing down the operation into the encodings can change the compressed type, from for instance the DDC type to a DDCFOR type on column 0. To change the type to FOR, we simply store an additional reference tuple with one value per column of the column group. This option does some operations depending on the type of input column, however most columns are able to change into a FOR version making the operations similarly scale in the number of column inputs. Worst-case scaling is $\mathcal{O}(\sum_{i=1}^{|\mathcal{G}|} d_i g_{ic})$.

The Effect: The benefit of these two tricks is that scalar operations reduce operation time from in the size of the dictionary’s unique tuples (times columns) $\mathcal{O}(\sum_{i=1}^{|\mathcal{G}|} d_i g_{ic})$ to scaling only in the number of columns $\mathcal{O}(\sum_{i=1}^{|\mathcal{G}|} g_{ic})$ equal to $\mathcal{O}(m)$ if not in overlapping state. However, the downside is if subsequent operations are not necessarily supported on FOR types of columns, but as seen later, it is not a problem once we introduce morphing compression to cheaply change between encodings. The subsequent sections describe in detail the individual operations, and how to efficiently execute them on encoded formats.

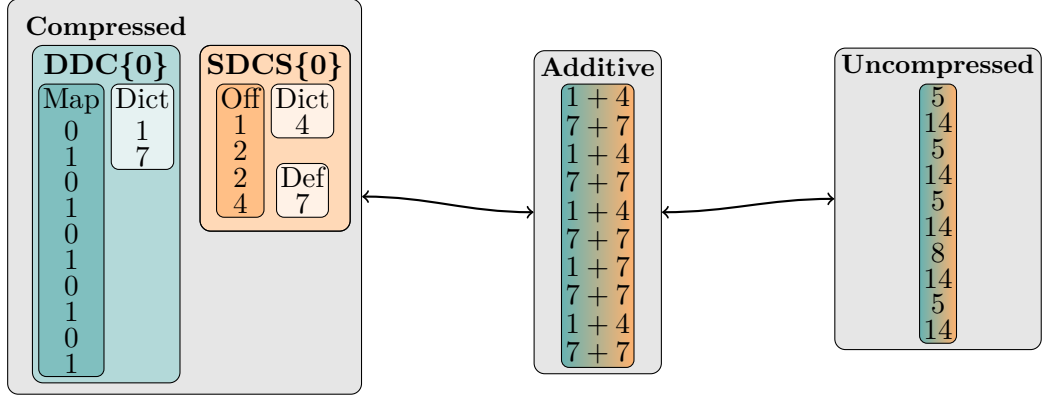


Figure 3.5: Overlapping Column Group Example

3.2.2 Overlapping Column Groups

After specific operations have to deal with—but can also leverage and propagate—the overlapping state with partial sum semantics. Figure 3.5 shows an example of two groups, and how they decompress additively. We implemented all column groups such that they always decompress additively to the output matrix. The additive decomposition means, in practice, that the column groups do not need to contain information about being in an overlapping state and can process operations pushed to them with no modifications to code. However, the methods calling into a compressed matrix block with overlapping column groups have to in most cases modify their behavior.

Operations With Overlapping Output Cells: Many overlapping operations fall into a category of overlapping output cells where multiple column groups’ output aggregate into the same uncompressed cells. Full or column aggregation like sum or mean therefore have to maintain thread-local results or locks on output matrices. Similarly, LMM is implemented and TSMM also work with careful assignment to outputs.

Operations Supported on Overlapping State: There are many operations, where the overlapping state that does not need modifications including matrix-scalar or matrix/row-vector multiplications. However, once the compressed matrix is in an overlapping state we can only push down linear binary operations. All other operations require decompression. The reason linear operations work is because of the associative property $((a + b) + c = a + (b + c))$ and distributive property $(a \cdot (b + c) = ab + ac)$ of linear operations. In essence, we can perform a scalar multiplication by the distributive property by pushing down the operation on all column groups. Similarly, we can perform scalar addition by adding an overlapping column group to the stack of overlapping groups because of the associative property.

Fully Decompressing: When some operations that are supported in compressed space, does not work in overlapping. All non-linear operations require decompression, such as $\text{pow}(\mathbf{X}^2)$, $\text{ReLU}()$ and $\text{abs}()$. Comparison between values, such as $<$, also break therefore also aggregations like $\text{max}(\mathbf{X})$, $\text{colMins}(\mathbf{X})$ where values also have to be compared decompress. In other words, operations like min/max/pow or matrix-matrix operations are not supported in overlapping state because these types do not distribute over sums. More critically, commonly-used activation functions contain some non-continuous functions, that require decompression. For the most used of these operations, we have implemented variations that fuse the decompression into an output matrix and in-place operations on the output matrix.

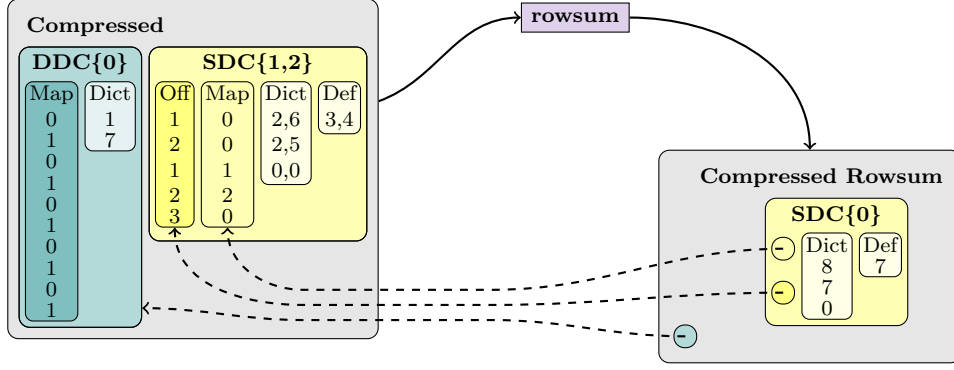


Figure 3.6: Example of AWARE Overlapping Rowsum.

3.2.3 Aggregations on Compressed Data

Aggregations on lightweight compression schemas is probably the most well-studied operation, since it is very simple and effective in the domain of DBMS systems. Aggregations like $\text{sum}(\mathcal{G}_i)$ or $\text{colSums}(\mathcal{G}_i)$ pre-aggregate counts and then scale and accumulate the dictionary \mathbf{D}_i . In CLA, column group types like OLE only need to aggregate segment sizes, but DDC column groups still required a full scan of the index structures. In AWARE, the most common column groups are DDC and SDC. Table 3.3 shows the asymptotic operational performance of performing sum and max on a compressed matrix. Memorizing the tuple frequencies, \mathcal{F}_i , of each unique value in the dictionary, \mathcal{D}_i , changes the asymptotic behavior of sum and mean removing the need to iterate through all rows first to collect a count. Instead, the compressed operations simply loop through the dictionary and sums dictionary tuples one at a time, to then multiply with a frequency and add to the global sum. The compressed sum operation of a single column group can therefore be defined as:

$$\text{sum}(\mathcal{G}) = \sum_{i=1}^d \mathcal{F}_i \cdot \sum_{j=1}^{\mathcal{G}_c} \mathcal{D}_{i,j} = \langle \mathcal{F}, \text{rowsum}(\mathcal{D}) \rangle$$

Asymptotic Change: Maintaining the counts changes the runtime from $\mathcal{O}(\mathcal{G}_{id}\mathcal{G}_{ic} + n)$ to $\mathcal{O}(\mathcal{G}_{id}\mathcal{G}_{ic})$. Aggregating operations such as max or min, do not need the tuple frequencies and therefore already perform efficiently similar to sum, $\mathcal{O}(\mathcal{G}_{id}\mathcal{G}_{ic})$.

Column Aggregation: Aggregating over the columns is similarly improving its performance if we maintain the count. The main difference from full aggregation is maintaining which columns each encoding aggregates into. Another difference only for sum is that we multiply directly each value instead of after summing a dictionary tuple.

Row Aggregation: Row aggregates are challenging because they are perpendicular to the column-encoded groups. In the min and max operations, we have to loop over all column

Table 3.3: Linear Algebra Asymptotic Runtime of Single Column Groups. This Table Shortens Column Group \mathcal{G}_i to \mathcal{G} and use $m = \mathcal{G}_{ic}$. \mathcal{FDC} is Fused Decompression and Operation. Table 2.2 Contains Missing Definitions.

Algebra Type	sum(\mathcal{G})			max(\mathcal{G})			$\mathcal{G} + x$				$\mathcal{G} \cdot x$			
	all	col	row	all	col	row	c	rv	cv	M	c	rv	cv	M
DDC	$\mathcal{O}(dm)$			$\mathcal{O}(dm)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(dm)$	or	$\mathcal{O}(m)$	\mathcal{FDC}	$\mathcal{O}(dm)$	\mathcal{FDC}		
SDC	$\mathcal{O}(dm)$			$\mathcal{O}(dm)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(dm)$	or	$\mathcal{O}(m)$	\mathcal{FDC}	$\mathcal{O}(dm)$	\mathcal{FDC}		
SDCZ	$\mathcal{O}(dm)$			$\mathcal{O}(dm)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(dm)$	or	$\mathcal{O}(m)$	\mathcal{FDC}	$\mathcal{O}(dm)$	\mathcal{FDC}		
SDCS	$\mathcal{O}(m)$			$\mathcal{O}(m)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(m)$		\mathcal{FDC}		$\mathcal{O}(m)$	\mathcal{FDC}		
SDCSZ	$\mathcal{O}(m)$			$\mathcal{O}(m)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(m)$		\mathcal{FDC}		$\mathcal{O}(m)$	\mathcal{FDC}		
CON	$\mathcal{O}(m)$			$\mathcal{O}(m)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(m)$		\mathcal{FDC}		$\mathcal{O}(m)$	\mathcal{FDC}		
FOR	$\mathcal{O}(dm)$			$\mathcal{O}(dm)$	$\mathcal{O}(n + dm)$		$\mathcal{O}(dm)$	or	$\mathcal{O}(m)$	\mathcal{FDC}	$\mathcal{O}(dm)$	\mathcal{FDC}		

Table 3.4: Binary Operations Scalar or Row-vector Change of Encoding

Input	Plus/Minus	Multiply/Divide	Modulus/Power
DDC	DDCFOR	DDC	DDC
DDCFOR	DDCFOR	DDCFOR	DDCFOR
SDC	SDCFOR	SDC	SDC
SDCFOR	SDCFOR	SDCFOR	SDCFOR
SDCZ	SDCFOR	SDCZ	SDCZ
SDCS	SDCS	SDCS	SDCS
SDCSZ	SDCS	SDCSZ	SDCSZ
EMPTY	CONSTANT	EMPTY	EMPTY
CONSTANT	CONSTANT	CONSTANT	CONSTANT
UC	UC	UC	UC

groups and find the minimum across encodings, making the execution scale in the number of column groups in the input times the number of rows $\mathcal{O}(n \sum_{i=1}^{|G|} d_i \mathcal{G}_{ic})$. There is one exception, if we only have one column group for the entire matrix, then we can simply aggregate the dictionary. For sum, it is different because we can leverage the overlapping concept. If there is few column groups compared to number of columns, it makes sense to make an overlapping row sum. Overlapping rowsum performs a row sum on each dictionary of each column group and returns an overlapping compression containing all the initial column groups index structures, but with rowsummed dictionaries, and column indexes all set to zero. Additive decompression of the overlapping result then is equivalent to the uncompressed rowsum.

Cumulative Sum: While not implemented, the column cumulative sum operation also fits nicely with AWARE. Cumulative sum is found many places in Linear algebra workloads. In compressed space if, for instance, the input column is a DDC encoded group, then we can simply return the DDC mapping and dictionary, and change it to a Delta DDC group. When decompressing the delta group, we would maintain the cumulative values so far, and add them to the current decompressing row. However, we have left this to future work, since we did not see a concrete application yet.

3.2.4 Binary Element-wise Operations

For matrix-scalar and matrix/row-vector operations—as used for standardization (e.g., $\mathbf{X} - \text{colMeans}(\mathbf{X})$)—we further preserve the structural redundancy by handling default values in SDC column groups (e.g., replace zero by column mean), leaving the index structures unchanged. The AWARE compressed format supports comparative operations (\leq , $=$, \geq , $<$, $>$, $!=$), multiply, addition, power, subtraction, modulus and bitwise operations.

Scalar and Row-vectors: We showed an example of scalar subtraction in Figure 3.4. Operations with row vectors can similarly be processed on the compressed representation without decompression by pushing the operations down into the individual column groups. The only contract each group have to adhere to is to not reallocate their index structures but only modify their dictionaries and/or reference values. Plus and Minus have the added ability to choose if we want to add the overlapping group, instead of applying to the column groups. Other operations, such as multiply, always have to be pushed down, because the overlapping concept is an additive one, not multiplicative. This approach is fundamentally different from the other compressed linear algebra frameworks that rely on sparsity, TOC [147], GLA [77], CSER [241], and sHAM [157], because we can exploit the original sparsity/redundancy through densifying operations such as binary plus. Table 3.4 shows how individual column groups can change the encoding type based on the operations performed. Notably, the densifying operations in many cases change the encoding into a FOR type. While some operations can push down into the dictionaries, keeping them as reference values of FOR is cheaper, and subsequent operations can then choose if it is more efficient to morph the encoding schemes.

Column-vectors or Matrices: If the other side of the binary operation is a full matrix or a column vector, then we have to decompress. The decompression happens because the compressed representation’s index structure \mathcal{I}_i does not align with the uncompressed vector or matrix input. Luckily, we do not have to decompress our compressed matrix fully to then do the operation, instead we fuse the binary operator with the decompression.

Fused Decompression: We implemented an fused decompression in cases where the operations are not supported, or we perform a non-linear operation on an overlapping compressed representation. The fused operation decompresses small row blocks in parallel into the output matrix, and applies the binary operator in-place on decompressed blocks.

3.2.5 Slicing

Linear algebra range selections, also called slicing, returns a given row and/or column range from a matrix. Slicing translates nicely to compressed operations. However, slicing does leave choices in returning compressed or uncompressed results depending on the size of the slice. Normal uncompressed slicing’s asymptotic behavior scales in the number of cells extracted $\mathcal{O}(r_n r_m)$, where r_n is the range length in rows and r_m is range length in columns.

Row Slices: Selecting full continuous row sections from the compressed data is fairly cheap, with worst cases $\mathcal{O}(r_n |\mathcal{G}|)$. The row slicing is done by slicing the index structures and keeping a pointer to the input’s dictionaries. We use this approach to tile compressed matrices for I/O, broadcasting to Spark. However, if the slice is small compared to the dictionary, then the compressed output could be substantially bigger than the uncompressed version. In such cases, we selectively decompress the selected range directly into an uncompressed output matrix, without first slicing the index structures.

Column Slices: Full column slices, on the other hand, do not need to edit index structures at all and performs even better, because of the column-oriented compression scheme. These properties translate to only editing the column indexes, and quickly removing column groups that do not have a column index within the selected range. We can quickly filter because the column indexes stored in the column groups is guaranteed sorted, which is one of the requirements in the compression scheme. The asymptotic time is $\mathcal{O}(r_m)$, but it does not reflect the actual speed.

Inner Slices: For any inner slice, we decompose it into first a column slice, and then performing the row slice. The order allows us to reuse the code for both of the above implementations. Slicing continuous ranges of rows from compressed frames and matrices is important for mini-batch algorithms, pre-processing, and writing. Additionally, data-centric ML pipelines often sample or select rows from input datasets via so-called selection matrix multiplications. In order to support these operations.

3.2.6 Matrix Multiplication

CLA [74, 75, 73] supports only matrix-vector and vector-matrix multiplications directly on compressed representations, but emulates matrix-matrix multiplications via repeated slicing and matrix-vector multiplications. This approach provides simplicity and reasonable performance, but looses performance as the size of the second matrix increases, which is common in applications like multi-class classification, dimensionality reduction, and clustering. Other previous work like TOC [147] supports matrix-matrix multiplication. However, they rely on a single encoding for the entire matrix, or mini-batch. Similarly the other compressed linear algebra (GLA [77], CSER [241]) rely on a single global encoding. Furthermore, many of them do not efficiently process dense compressed matrices. In this section, we introduce simple yet impactful techniques for matrix-matrix multiplications on lightweight compressed matrices, including special cases of compressed-compressed multiplication.

Pre-aggregation: A central technique of compressed matrix multiplications in AWARE and CLA are different forms of pre-aggregation over the distinct tuples. In AWARE, we vectorize

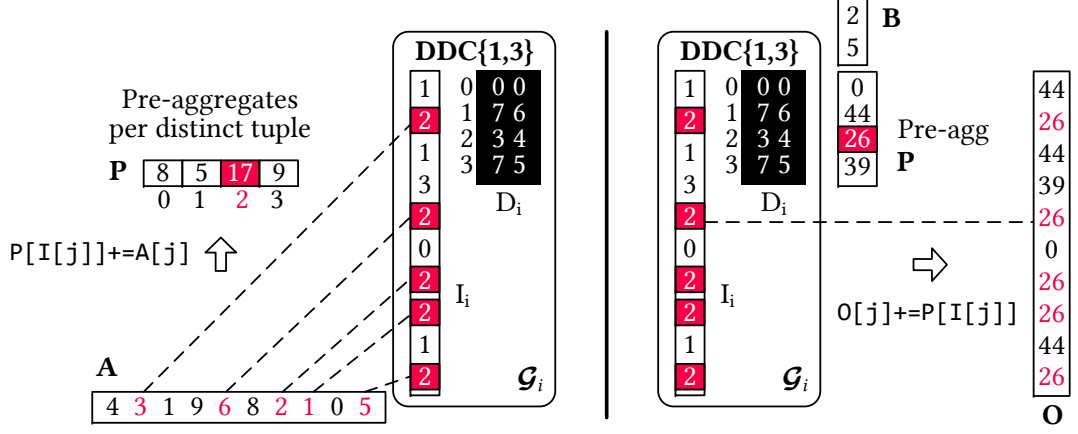


Figure 3.7: Example Pre-aggregation in Compressed Matrix Multiplication. Left is for LLM and right is RMM.

such pre-aggregation for improved simplicity and performance. For instance, Figure 3.7 shows the intuition of pre-aggregation in left and right matrix multiplication. First, for a left matrix multiplication $\mathbf{A} \mathcal{G}_i$ with an uncompressed vector \mathbf{A} , we initialize a zero vector \mathbf{P}_i , and accrue the entries of \mathbf{A} according to indexes \mathcal{I}_i . Subsequently, a simple uncompressed vector-matrix multiplication $\mathbf{P}_i \mathcal{D}_i$ of the pre-aggregates and the dictionary yields the overall result (for columns of the column group). Instead of multiplying all entries with the same distinct value (or tuple in case of co-coding), we simply distribute multiplication over addition. Second, for a right matrix multiplication $\mathcal{G}_i \mathbf{B}$ with an uncompressed vector \mathbf{B} (subset relevant to the column group), we first compute a matrix-vector multiplication of $\mathcal{D}_i \mathbf{B}$ to get the pre-aggregated vector \mathbf{P}_i . If we stop the right matrix multiplication there, and simply keep the \mathbf{P} as a new dictionary it becomes an overlapping matrix. We can simply decompress to add these pre-aggregated values to the output according to indexes \mathcal{I}_i . Interestingly, a similar pre-aggregation strategy is also applied as a general case for unnesting correlated subqueries [170]. Given this vectorized form and the storage of dictionaries as uncompressed matrices, we can directly apply cache-conscious uncompressed matrix multiplications for the general case of left- or right-hand-side uncompressed matrices with k rows or columns, respectively. Dense pre-aggregation, such as in DDC, have complexity in number of rows $\mathcal{O}(n)$. However, sparsity exploiting encodings such as SDCZ can improve it to $\mathcal{O}(nnz(\mathcal{G}_i))$.

3.2.6.1 Left Matrix Multiplication

We call the matrix multiplication $\mathbf{A} @ \mathcal{G}$ —where the left-hand-side input \mathbf{A} is uncompressed—a left matrix multiplication (LMM). Figure 3.8 shows an LMM with two column groups, color-coded blue with dots and green filled. For each column group, we compute the pre-aggregated $k \times d_i$ matrix \mathcal{P}_i (via the already described vectorized pre-aggregation), then matrix multiply $\mathcal{P}_i @ \mathcal{D}_i$, and finally, shift these results into the correct column positions of the output matrix

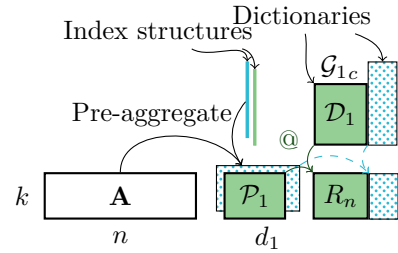


Figure 3.8: Left Matrix Multiply ($\mathbf{A} @ \mathcal{G}$).

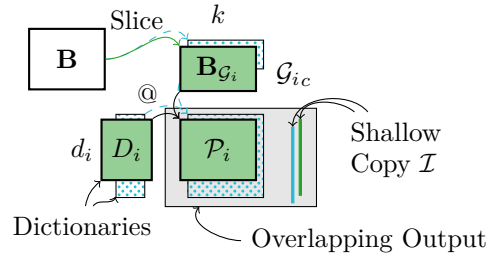
\mathbf{R} . Pre-aggregation for each column group is a scan of \mathbf{A} , but for large index structures \mathcal{I}_i , we can utilize cache-blocking to reuse blocks of \mathcal{I}_i from caches across multiple rows in \mathbf{A} . The more co-coding is applied, and/or the smaller the number of distinct items per group, the more we benefit from LMM pre-aggregation in terms of reduced floating point operations and data accesses. Multi-threaded LMM operations parallelize over column groups and rows in \mathbf{A} because they access disjoint output columns.

Algorithm 4 Left Matrix Multiply With Pre-morphing**Require:** Matrix input \mathbf{A} , Column Groups \mathcal{G} **return** R $R \leftarrow \mathbb{R}^{\mathbf{A}_r \times \mathcal{G}_c}$, $\text{constV} \leftarrow \mathbb{R}^{R_c}$, $\text{PreAggGroups} \leftarrow \{\}$ **for** $g \in \mathcal{G}$ **do** \triangleright Loop through all column groups **if** g instanceof MorphingGroup **then** $g_m = g.\text{extractCommon}(\text{constV})$ \triangleright Sparsify morphing groups $\text{PreAggGroups.add}(g_m)$ \triangleright Add morphed groups to groups to perform LLM **else if** g instanceof Const **then** $g.\text{extractCommon}(\text{constV})$ \triangleright Constant groups can be removed from LLM **else** $\text{PreAggGroups.add}(g)$ \triangleright If modifications are unnessesary **end if****end for** $\text{rowSums} \leftarrow \text{rowSum}(\mathbf{A})$ \triangleright Calculate the rowSums of L**for** $g_m \in \text{PreAggGroups}$ **do** $R \leftarrow R + \text{LMM}(\mathbf{A}, g_m)$ \triangleright Add the output of each column group**end for** $R \leftarrow R + \text{rowSums} \otimes \text{constV}$ \triangleright Add outer product of row sums and constantV

Pre-Morphing: As an optional preprocessing step we can cheaply morph column groups that would perform better in other closely related formats, and extract a common multiplication vector that we can multiply with a rowsum of the left matrix. Algorithm 4 shows the pre-morphing in an algorithm environment. First, we allocate the standard output of the matrix multiply, and then we loop though all groups to extract groups that benefit from morphing. As an example SDC benefits, because it has a default value that in normal instances would mean each value on the left matrix would have to be processed. While on the other hand SDCZ is able to skip all the common elements. To morph the SDC into SDCZ is simply by subtracting the common default value of SDC from all tuples in the dictionary, and adding the default value to the commonV vector. Similarly SDCS morph to SDCSZ, DDCFOR to DDC, and SDCFOR to SDCZ. Most of the morphing groups only modify values in the size of their individual number of columns, making the modification very efficient and cheap. With the constV extracted from the morphing groups, we can simply add a correction to the LLM result by adding the dot product of the row sum of the left matrix and the constV to the result matrix.

3.2.6.2 Right Matrix Multiplication

Similar to LMM, we call a matrix multiplication ($\mathcal{G} @ \mathbf{B}$)—where the right-hand-side input \mathbf{B} is uncompressed—a right matrix multiplication (RMM). Figure 3.9 shows an example with two column groups. Our column-oriented compression and multiplication by \mathbf{B} from the right, provides an opportunity to preserve structural redundancy and thus, avoid unnecessary decompression (aggregation into an uncompressed output). The simple, yet very effective, key idea of our RMM is to only perform the vectorized pre-aggregation $\mathcal{P}_i = \mathcal{D}_i @ \mathbf{B}_{\mathcal{G}_i}$ by multiplying the column group dictionaries with related rows in \mathbf{B} , and then store these pre-aggregates as new dictionaries of overlapping column groups. This way, we can leave the index structures \mathcal{I} untouched and shallow-copy them into the compressed output representation, preserving the source redundancy. Each output column group now has dictionaries of size $d_i \times k$. The individual column groups compute, again

Figure 3.9: Right Matrix Multiply ($\mathcal{G} @ \mathbf{B}$)

independent (but now overlapping) outputs and thus, multi-threaded operations parallelize over column groups and columns of \mathbf{B} . In distributed environments with block-local matrix multiplications, the same RMM applies and overlapping outputs can be preserved (if beneficial in size) through serialization and shuffling.

Pre-Morphing: Similarly to LLM, RMM benefits from morphing the column groups. We allocate the constant vector and morph the groups to improve sparsity exploitation. The main difference is that the vector is fully matrix multiplied with the right side matrix and added as another overlapping column group to the output. In cases of recurrent multiplications the overlapping constant vector group is reused and the number of groups do not grow.

3.2.6.3 Self Matrix Multiplication

Transpose-self matrix multiplication (TSMM)³ $\mathbf{X}^\top \mathbf{X}$ or $\mathbf{X} \mathbf{X}^\top$ —whose outputs is known to be symmetric—appears in many applications such as closed-form linear regression, co-variance and correlation matrices, PCA, and distance computations. We only implemented the inner TSMM, $\mathbf{X}^\top \mathbf{X}$, most commonly used. CLA emulates TSMM again via slicing and repeated vector-matrix multiplications. In contrast, AWARE

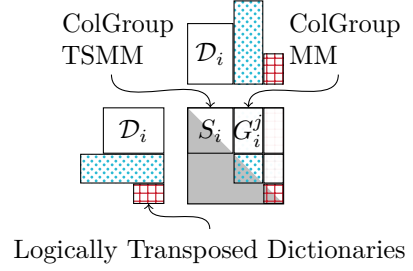


Figure 3.10: Transpose-Self MM($\mathcal{G}^\top \mathcal{G}$)

natively supports TSMM as shown in Figure 3.10, as well as compressed-compressed matrix multiplications (with transposed left input), which are also commonly occurring in practice. A TSMM is composed of pairs of column group operations, where blocks on the diagonal are self-joins of column groups, while others are $|\mathcal{G}| \cdot (|\mathcal{G}| - 1)/2$ combinations.

Self TSMM: A self-multiplication of a column group computes \mathcal{S}_i by using the tuple frequencies of the column group \mathcal{F}_i , and computes the results via a specialized uncompressed TSMM of \mathcal{D}_i scaled by \mathcal{F}_i on one side. We have implemented a specialization for dense or sparse \mathcal{D}_i that multiplies the tuple frequencies in the TSMM loops. We further only calculate the upper triangle of the TSMM diagonal blocks, and add its values directly to the output matrix with no intermediate allocations. Overall, this implementation makes the diagonal self blocks very efficient to compute scaling according to $\mathcal{O}(\mathcal{G}_{ic}(\mathcal{G}_{id})^2)$.

ColGroup TSMM: The remaining blocks adopt the strategy of LMM: pre-aggregation and matrix multiplication $\mathbf{P}_i \mathcal{D}_i$. However, given two compressed inputs, we can freely pick the pre-aggregation side and alternatively, do $(\mathbf{P}_j \mathcal{D}_j)^\top$. In any case, the pre-aggregate is computed without decompression and can exploit column-group characteristics (e.g., sparse, non-default, or constant encodings).

Pre-Morphing: Again, similar to both RMM and LMM, we can pre-morph the column groups to exploit more sparsity in the encodings, but in the TSMM case it is a bit more complicated to add the correction. First, we calculate the column sum of the compressed matrix, leveraging the efficient compressed column sum. Then, we can add the outer product of the column sum and the constant vector extracted while morphing. However, to produce correct results two other outer products have to be added with scaling in the number of rows on the extracted column sum to take into consideration the extracted correction. Algorithm 5 shows how all three dot products can be fused into two loops, only calculating the top half.

Finalize: Finally, before returning the top diagonal of the matrix is copied down to the bottom half. Only calculating the top halves the compute time of TSMM.

3.2.6.4 Compressed Matrix Multiplication

Currently, we support compressed matrix multiplication in all cases, with fallback to decompression of one side if there is no specialized method for performing the full multiplication

³TSMM is also known as BLAS syr (symmetric rank-k update) or Gram matrix.

Algorithm 5 TSMM Correcting Dot Product

Require: Matrix output R , $constV \leftarrow \mathbb{R}^{R_c}$, $colsum \leftarrow colsum(PreAggGroups)$

return R

for $row = 0$ to n **do**

$off \leftarrow rows \cdot row$ ▷ assuming a dense linearized output R

$v1 \leftarrow constV[row]$ ▷ Dot product 1

$v2 \leftarrow colsum[row] + constV[row] * nRow$ ▷ Fused dot product 2

for $col = row$ to n **do** ▷ Only top half of the matrix

$corection \leftarrow v1 \cdot colsum[col] + v2 \cdot constV[col]$ ▷ Other half of dot products

$R[off + col] \leftarrow R[off + col] + corection$

end for

end for

compressed. In general, the rule is to try to perform an RMM type of multiplication if possible because of the better characteristics of constructing an compressed overlapping output. However, we do support a specialized LLM version of $C = A^t B$ (A and B is compressed), because the output tends to be a small uncompressed matrix.

Specialized TCLMM: Algorithm 6 shows the double compressed left transposed matrix multiplication. Similar to TSMM, it utilize the column group multiplication, just on every column group instead of only the upper triangle. This multiplication also leverages the pre-morphing of column groups to improve the operations performance. To correct for the pre-morphing, we perform three dot products afterwards similar to TSMM however with some differences. First, we have to get the column sum of either compressed matrix input. Computing column sums is, as stated before in Section 3.2.3, very cheap in compressed spaces.

Parallelization: Parallelizing the double compressed multiplication can be a bit tricky. We have tried making a task of each column group multiplication, however that parallelization was too fine-grained, and many cores would have to load different column groups more or less randomly. In the end, we settled for a row group per task scheme, such that each inner loop in Algorithm 6 is run in parallel. Notably, it is only possible to perform the operation in parallel if there are no overlapping column groups, otherwise each thread would add into the same output cells. The final correcting dot products have to be done after the full multiplication, or selectively together with non overlapping column groups ranges once their tasks are finished. A future work in the compressed multiplication include fuse all the last level dot products into a specialized kernel similar to a matrix multiplication. This fused operator could then be executed in parallel over blocks.

Algorithm 6 Left Transposed Compressed Compressed Matrix Multiply

Require: Column Groups left \mathcal{G}^l , Column Groups right \mathcal{G}^r

return R

$R \leftarrow \mathbb{R}^{\mathcal{G}_c^l \times \mathcal{G}_c^r}$

$PreAggGroups^l, constV^l \leftarrow \text{prefilter}(\mathcal{G}^l)$

$PreAggGroups^r, constV^r \leftarrow \text{prefilter}(\mathcal{G}^r)$

$colSums^l \leftarrow colSum(PreAggGroups^l)$

$colSums^r \leftarrow colSum(PreAggGroups^r)$ ▷ Each colsum scale in $\mathcal{O}(\sum_{i=1}^{|\mathcal{G}|} d_i \mathcal{G}_{ic})$

for $g_m^l \in PreAggGroups^l$ **do**

for $g_m^r \in PreAggGroups^r$ **do**

$R \leftarrow R + \text{leftMultColGroups}(g_m^l, g_m^r)$

end for

end for

$R \leftarrow R + constV^l \otimes (constV^r \cdot \mathcal{G}_r^l)$ ▷ Dot product scaled by rows

$R \leftarrow R + constV^l \otimes colSums^r$ ▷ Two other correcting dot products

$R \leftarrow R + colSums^l \otimes constV^r$ ▷ Each dot product is $\mathcal{O}(\mathcal{G}_c^r \mathcal{G}_c^l)$

Table 3.5: Complexity of Compressed Matrix Multiplications.

	Uncompressed (dense)	AWARE (multiple column groups)
LMM	$\mathcal{O}(knm)$	$\mathcal{O}\left(k(\sum_{i=1}^{ \mathcal{G} }(\mathcal{G}_{ic}(\mathcal{G}_{i\mu}^c + d_i)))\right)$
RMM	$\mathcal{O}(nmk)$	$\mathcal{O}\left(k\sum_{i=1}^{ \mathcal{G} }(d_i\mathcal{G}_{ic})\right)$
TSM	$\mathcal{O}(nm^2)$	$\mathcal{O}\left(\sum_{i=1}^{ \mathcal{G} }(\mathcal{G}_{ic}(d_i\mathcal{G}_{ic} + \sum_{j=i+1}^{ \mathcal{G} }(\mathcal{G}_{j\mu}^c + d_j\mathcal{G}_{jc})))\right)$
TCLMM	$\mathcal{O}(m_1nm_2)$	$\mathcal{O}\left(\sum_{i=1}^{ \mathcal{G} }(\mathcal{G}_{ic}\sum_{j=1}^{ \mathcal{G} }(\mathcal{G}_{j\mu}^c + d_j\mathcal{G}_{jc})))\right)$
SLMM	(sparse) $\mathcal{O}(km)$	$\mathcal{O}(km)$

3.2.6.5 Selective Matrix Multiplication

Another type of matrix multiplication we added a specialization for in BWARE is selective matrix multiplication (SLMM). Selecting a random sample from a matrix can be done via matrix multiplication. If we define a sparse matrix \mathbf{S} , with a single 1 in each row and then left matrix multiply it with a matrix \mathbf{X} , this operation selects rows from \mathbf{X} . A 1's position in \mathbf{S} determines the source row by its column position and the target row by its row position. This multiplication is, for instance, used in random reshuffling of a matrix before SGD. For compressed selection multiplication, we use a left compressed matrix multiply that does not pre-aggregate the intermediate matrix, unlike GCM [77], TOC [147], AWARE [23], and CLA [74]. Instead, we use the non-zero values of the left side matrix (guaranteed to be all 1) to selectively extract compressed row tuples, and decompress them into the output matrix. This solution leverages the index structures of the compression schemes.

Future Work: We did not implement a version that returns a compressed matrix, because we only had use cases where the selection in the workloads was very small, e.g., cluster initialization points. Therefore best for performance of subsequent operations was an uncompressed output. However, if one want to randomly sample *large samples* from a compressed matrix, it would make sense to implement a version that would extract index structures from the column groups and return compressed matrices instead.

3.2.6.6 Matrix Multiplication Asymptotic Runtime

The asymptotic runtime of the compressed multiplications can be a bit difficult to quantify. In detail, Table 3.5 compares the asymptotic behavior of uncompressed matrix multiplications with related AWARE operations. Uncompressed LMM and RMM have both cubic complexity, while sparse linear algebra can improve by computing only non zero elements on either side, at an increased cost to each operation due to indirections though sparse index structures. In contrast, compressed LMM and RMM have a complexity has depends on the data characteristics (distinct items and co-coding per column group). To further complicate the analysis, normal multiplication does multiplications and additions together. While the compressed matrix multiplication have different phases, where pre-aggregation only use additions and dictionary-based matrix multiplication use a standard matrix multiplication.

LLM Cases: With substantial co-coding (e.g., $|\mathcal{G}| = 1 \wedge \mathcal{G}_{ic} = m$) and few distinct items $d_i \ll n$, a much better complexity than the uncompressed is possible ($\mathcal{O}(k(nnd(\mathcal{G}_i)) + kd_im)$ instead of $\mathcal{O}(knm)$). In the worst-case, $|\mathcal{G}| = m$ and $d_i = n$, gives a compressed asymptotic runtime equal to uncompressed, but with a higher constant factor.

RMM Cases: Because RMM does not iterate through \mathcal{I} it has a lower asymptotic behavior than LMM, and thus, we already benefit with less favorable data characteristics the same worst-case guarantees apply, even with subsequent decompression.

TSM Cases: TSM is more complex, but the first term in the outer sum operation represents self-joins per column group (including pre-aggregation). These self joins scales $\mathcal{O}(\mathcal{G}_{ic}^2 d_i)$ for each column group but they only have to calculate half of their specific output matrix, and they are very cache-friendly. The second term enumerates pairs of column groups

with two sub-terms for pre-aggregation and scaling, each column group multiplication scaling according to $\mathcal{O}(\mathcal{G}_{ic}(nnd(\mathcal{G}_j) + d_j\mathcal{G}_{jc}))$. The best case for TSMM is similar to LMM with $|\mathcal{G}| = 1 \wedge \mathcal{G}_{ic} = m$ removing the need for column group multiplications reducing the runtime to $\mathcal{O}(md_im)$. The worst case increase the column group multiplications, that introduce overheads.

TCLMM Cases: The final compressed-compressed multiplication similarly benefit from few column groups with many columns, reducing to a similar to LLM $\mathcal{O}(\mathcal{O}(k(nnd(\mathcal{G}_i)) + kd_im))$. However, with one major distinction, we can chose what side to pre-aggregate, and therefore, leverage the best compressed side.

3.2.7 Decompression

Unsupported, not implemented, or operations that cannot be processed in the compressed state of AWARE are handled via decompression. We use cache-conscious blocking for converting the column-compressed matrix into either a row-major dense uncompressed matrix or a sparse MCSR matrix. No matter which compressed state we are in, we decompress row blocks in parallel for better cache efficiency.

Normal Decompression: The normal decompression decompresses at most one value per cell in the output matrix, making it scale linearly with the number of cells, $\mathcal{O}(nm)$. In sparse cases the number of values added into the output is less because we only add non-zero elements to the output. However, exploiting sparsity requires either the encoding to be SDCZ or a sparse dictionary in DDC or SDC. Similarly to the compressed matrix multiplications, it can be beneficial to extract the common values from column groups, but if we extract the common values, we only support decompressing into a dense matrix.

Overlapping Decompression: The overlapping decompression adds a term to the asymptotic time complexity $\mathcal{O}(nm|\mathcal{G}|)$ (worst-case of $|\mathcal{G}|$ overlapping groups). Therefore, we want the optimizer to reduce the number of column groups with more aggressive co-coding if we end up in a state with overlapping decompression. Although the overlapping decompression is expensive, the pre-morphing allows each group to not add into every cell in the output, making it essential in overlapping cases with many groups that can exploit $\mathcal{G}_{i\mu}^c$.

Partial Decompression: We also support partial decompressions to enable the operations to process sub-parts of the matrix at a time. Partial decompressions are also used for slicing operations, where if we slice sufficiently small blocks it is more efficient to just decompress those cells.

3.3 Workload-based Adaptation of Compression Plans

Apart from reduced I/O and memory-bandwidth requirements due to the smaller compressed size, compression can also reduce the number of floating point operations. Cost functions based on asymptotic behavior of each compression type, together with estimated or observed compression ratios, are the basis for our workload-aware compression planning, allowing us to optimize for total execution time. AWARE aims to achieve broad applicability by redundancy-exploitation and optimization for execution time. Instead of compressing input matrices only according to data characteristics, we extract workload characteristics from the given linear algebra program, and compress candidate inputs and intermediates in a data-*and* workload-aware manner, and then leverage compressed data characteristics for a refined compilation of execution plans.

3.3.1 Workload Trees

Given a linear algebra program, workload-aware compression selects intermediates as compression candidates, and for each candidate extracts a workload tree (a compact workload summary seen in [Figure 3.11](#)), evaluates its costs, and if valid for compression, injects a `compress` directive that utilizes the workload for fine-tuned (i.e., workload-aware) compression.

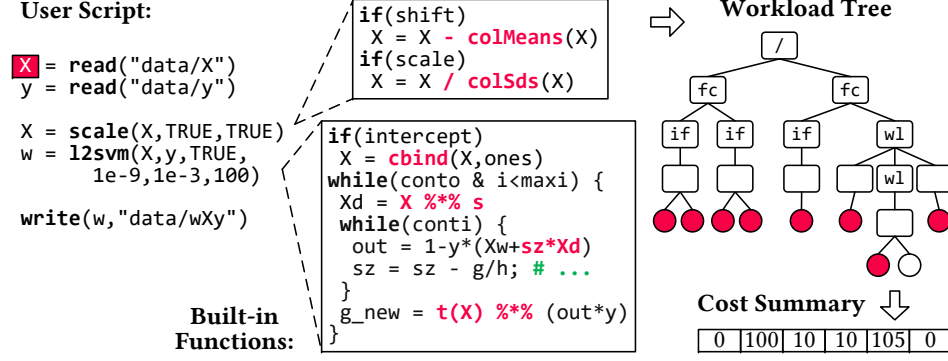


Figure 3.11: Example showing a user script that reads a feature matrix \mathbf{X} and label vector \mathbf{y} , normalizes \mathbf{X} to mean 0 and standard deviation 1, and then trains an l2-regularized support vector machine. These functions are themselves linear-algebra scripts. Assuming \mathbf{X} as a compression candidate, we extract the workload tree at the right, which contains 2 function calls, 3 if branches, 2 nested while loops and 8 compressed operations and 1 decompression. Aggregating the workload tree yields a cost summary for categories of operations.

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

Workload Tree Extraction: Our initial candidate selection and optimization approach relies on heuristics. We make a linear scan over the program, and extract compression candidates by operation type (e.g., persistent reads, comparisons, `ctable`, and rounding) as well as shape constraints (dimensions, and row/column ratios). Together, these heuristics find good candidates while keeping the number of candidates low. For each candidate, we then make a scan over the program and extract its workload tree by computing the transitive closure of derived compressed intermediates (based on operation types that are known to produce compressed outputs). Again, in a heuristic manner, we then evaluate individual candidates independently without considering joint effects of groups of compressed intermediates. This extraction also descends into functions, but via stack-based identification includes only the first level of recursive function calls. In this context, we prune the unnecessary extraction of workload trees for overlapping intermediates. We perform this extraction at the end of inter-procedural analysis (IPA). At this point, literals and size information have been propagated across the program and into amenable functions, and many simplifications have been performed. In Figure 3.11, we would have propagated the shift, scale, and intercept flags, removed unnecessary branches, and inlined the scale function into the main program.

Cost Evaluation: The workload vector summaries computed from the workload tree serve two purposes. First, to compare to uncompressed operations cost and second, to guide compression planning of individual column groups costs. The frequencies reflect the size differences of intermediates via multipliers in the counts. We organize the cost summaries by categories of operations with different behavior in compressed space: (0) Decompression, (1) Overlapping Decompression, (2) LMM, (3) RMM, (4) TSMM, (5) Dictionary-Ops, and (6)

Indexing-Ops. Decompression is the frequency of regular decompressions while overlapping decompression converts the overlapping output into uncompressed form if operations are not applicable on partial aggregates. LMM is multiplied by the number of rows on the left, RMM is multiplied by the number of columns on the right, and TSMM includes counts of compressed multiplications and transpose-self multiplications and is multiplied by the number of columns. Dictionary operations can be performed directly on the compressed dictionaries (e.g., sum or element-wise scalar operations). Finally, Indexing refers to the slicing of batches or blocking during broadcasting. The cost vector also contains a boolean set to true if there are densifying operations that would make an uncompressed input matrix dense somewhere in its workload tree. The boolean does not accurately reflect the number of operations performed on dense vs sparse intermediates. However, in densifying cases, the dense performance typically dominates execution time. If the workload vector suggests that compressing an intermediate may be beneficial, we make the cost summary globally available for subsequent runtime compression instructions and inject the related `compress` directive.

3.3.2 Compiler Integration

A challenge is that compression happens at runtime, and thus, the estimated and actual compressed size is unknown during initial compilation. Accordingly, we create—similar to data-dependent operators with unknown output shapes [32]—artificial recompilation opportunities by splitting basic blocks after injected `compress` directives. If a block contains multiple, independent `compress` operators, we create a single cut for all.

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

3.3.3 Workload-dependent Runtime Compression

The `compress` directives injected into the execution plan, perform compression as described in Section 3.1.3, but for workload-awareness get the workload vector summary as input. The workload vector influences the selection of column group types, co-coding decisions, and tuning for compression ratios. During classification and co-coding, we estimate the column costs from the sample, and then use these costs to decide on column groupings instead of grouping purely for compression ratios. However, including I/O costs also enables adapting the compression plans for large out-of-core datasets where good compression ratios are important to fit data in memory and/or reduce I/O. Local compression directly leverages the workload vector, while for distributed compression, we serialize the workload vector and compress blocks independently. In the context of hybrid runtime plans—composed of local in-memory, and distributed Spark operations—after compression, opportunities for compiling more efficient plans arise.

Table 3.6: Datasets (n Rows, m Columns, sp Sparsity).

Dataset	n (nrow(\mathbf{X}))	m (ncol(\mathbf{X}))	sp	Size
Airline78 [17]	14,462,943	29	0.54	3.4 GB
Amazon [171, 105]	8,026,324	2,330,066	1.2e-6	1.22 GB
Covtype [71]	581,012	54	0.22	127 MB
Mnist1m [43]	1,000,000	784	0.25	2.46 GB
US Census [71]	2,458,285	68 (378)	0.43 (0.18)	1.34 GB
US Census 128x	314,660,480	68 (378)	0.43 (0.18)	289.5 GB

3.4 Experiments

This section contains the experiments from the AWARE paper [23]. The experiments study AWARE in Apache SystemDS⁴ in comparison with uncompressed operations (ULA), compressed linear algebra (CLA) [74, 75] in Apache SystemML, and different data types in TensorFlow. We evaluate a variety of micro benchmarks, end-to-end ML algorithms, and hyper-parameter tuning; with local, distributed, and hybrid runtime plans.

3.4.1 Experimental Setting

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

Datasets: Since compression is strongly data-dependent, we exclusively use the real datasets shown in Table 3.6. This selection includes dense, sparse, and ultra-sparse datasets with common data characteristics. All reported sizes and compression ratios refer to the size in memory using a sparsity threshold of 0.4 for uncompressed matrices. US Census [71] is further used in an encoded form with binning/one-hot encoding for numerical, and recoding/one-hot encoding for categorical features, resulting in an increase from 68 to 378 columns, and the increased sparsity from 0.43 to 0.18, but with negligible change of the size in memory. For large-scale experiments, we use a replicated versions of US Census Enc (up to 128x) which is roughly 290 GB and after densifying operations more than 950 GB. The Spark default configuration uses a storage fraction of 0.5, which gives an aggregate cluster memory of $6 \cdot 105 \text{ GB} \cdot 0.5 = 315 \text{ GB}$. That way, we scale to data sizes that require I/O per iteration in uncompressed representation.

3.4.2 Compression Performance

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

Compression Ratios: Starting with local single-node matrix compression, we compare AWARE optimized for memory (AWARE-Mem) and for workload (AWARE) with the existing CLA framework [74, 75]. The used workload is a fixed cost summary of left matrix multiplications that leads to extensive co-coding of columns. Table 3.7 shows the compression times and ratios for all datasets, where the ratios are calculated from the sizes of in-memory

⁴All code and experiments are available open source in Apache SystemDS (<https://github.com/apache/systemds>) and our reproducibility repository (<https://github.com/damslab/reproducibility>).

representations. We attribute the minor differences to published CLA compression ratios [75] (Airline78 7.44, Covtype 18.19, US Census 35.69) to different data preparation and sparse memory estimates. Compared to CLA, AWARE yields worse compression ratios except for CovType, where AWARE’s co-coding finds other column groups and uses new encoding types. The lower compression ratios are due to the tuning for operations performance rather than size, and in practice, moderate absolute differences of large compression ratios have only little impact on size. For example, compressing a 1 GB input with ratio 7 versus 6 yields only a difference of 24 MB. The Amazon dataset is an interesting case, where CLA runs out of memory due to group memoization during co-coding ($> 2.7 \cdot 10^{12}$ pairs of columns). In contrast, AWARE aborts the compression early because the estimated total costs exceed the costs of ULA. Optimizing for memory in AWARE yields a low compression ratio for Amazon because of object overheads per column group, which do not exist in ULA’s CSR representation.

Compression Times: Table 3.7 further shows the compression times for all datasets. AWARE generally reduces compression times when optimizing for size (up to 8x) and cost (up to 3.7x), which make compression easier to amortize. The speed difference in AWARE-Mem US Census Enc is due to a reduction of the Grouping phase from 27 to 2.7 seconds. AWARE generally reduces compression times when optimizing for size (up to 4.7x) and cost (up to 4.2x), which makes compression easier to amortize. The speed difference in AWARE-Mem US Census Enc is due to a reduction of the grouping phase from 19 to 2.4 seconds. When using workload-aware compression with a fixed cost summary that causes more grouping and compression, only MNIST and US Census have significantly slower compression compared to optimizing for memory. MNIST is slow because combining column groups have a large number of distinct values (each column contains up to 256 distinct values, and three columns together has up to 256^3 distinct tuples).

Compression Ratio Spark: For distributed operations, matrices are represented as Spark resilient distributed datasets (RDD) [255]—i.e., distributed collections of key-value pairs—with values being fixed-size matrix blocks of size $b \times b$ (except boundary blocks). These blocks are compressed independently with separate compression plans and dictionaries. The default block size is $b = 1K$ (8 MB dense blocks), but sparsity and compression warrant larger blocks. Table 3.8 varies this block size b for US Census, and reports the size of AWARE-Mem, AWARE, and uncompressed (ULA) RDDs (from Spark’s cached RDD-infos). With small blocks, there is larger variability of compression, and increasing block sizes give better ratios while also stabilizing the resulting compression plans. Overall, AWARE yields good compression ratios even with small b and approaches local compression ratios with larger b . For the remaining experiments, we use a block size of $b = 16K$.

3.4.3 Operations

In a second series of micro-benchmarks, we compare the runtime of AWARE with ULA and CLA operations. While CLA was designed for operations performance close to uncompressed and benefits from keeping large datasets in memory, AWARE aims to improve performance more generally, even for in-memory settings and keep performance stable even if the input

Table 3.7: Local Compression Times [Seconds] and Ratios.

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

Table 3.8: Spark RDD Compression (Data: US Census Enc).

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

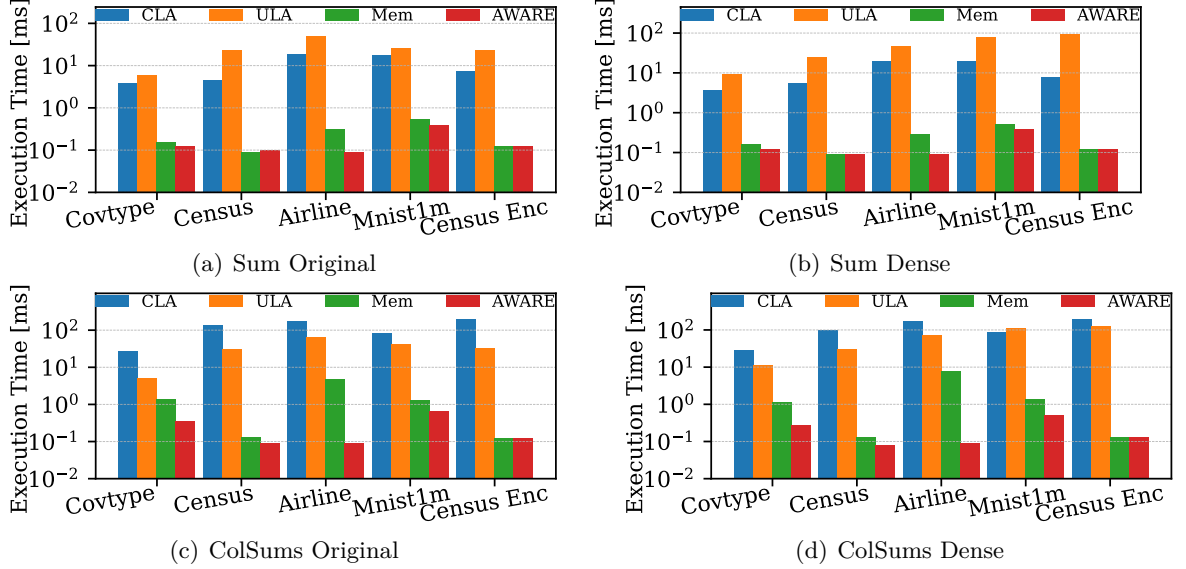


Figure 3.12: Operations Performance Aggregate.

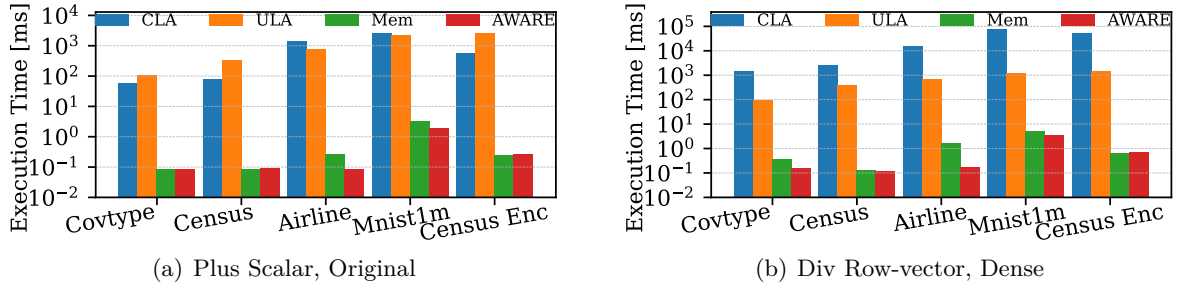


Figure 3.13: Operations Performance Scalar.

data is densified. We compare both variants of original and densified data where for the latter we simply add 1 to all cells.

Aggregations: Figure 3.12(a) shows the results for computing the aggregate sum(\mathbf{X}). CLA processes each column group in parallel, aggregates individual sums, and combines them into the result. ULA uses multi-threading with sequential scans of row partitions. By memorizing the frequencies of tuples, AWARE executes purely on the column group dictionaries without scanning their indexes because of memoization. Without memoization, the execution time increases according to the complexity of scanning the index structures. For instance, we observed no penalty in case of Census Enc, while 60x performance drop in Census for AWARE. When densifying, in Figure 3.12(b), CLA and AWARE retain their performance while ULA's performance worsen. For column aggregations (Figure 3.12(c) and Figure 3.12(d)), CLA is slower than ULA because CLA's DDC colSums is not specialized for DDC1 and DDC2 and thus, performs a lookup of dictionary values for each encoded cell.

Element-wise Operations: Figure 3.13(a) shows the performance of adding a scalar value to each matrix cell. We observe extreme speedups of up to 10,766x because AWARE

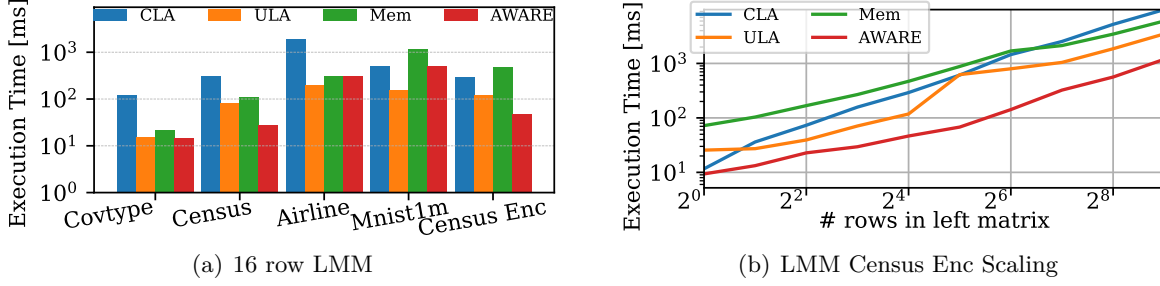


Figure 3.14: Operations Performance Left Matrix Multiplication.

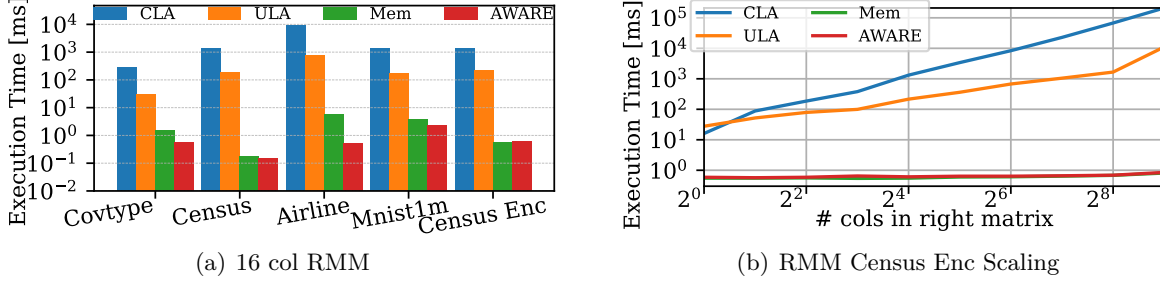


Figure 3.15: Operations Performance Right Matrix Multiplication.

avoids modifying dictionaries where possible. Figure 3.13(b) shows similar improvements for matrix-vector (row vector) element-wise operations. Specifically, we analyze \mathbf{X}/\mathbf{v} on sparse (not shown) and dense representations, where we chose division because it forces modifications of the dictionaries. We still see speedups of about three orders of magnitude (2,047x).

Left Matrix Multiplication (LMM): Left, right, and transpose-self matrix multiplications are key operations in many ML algorithms. In the following, we first evaluate these operations independently. Figure 3.14(a) shows the results of left matrix multiplications for all datasets, where the uncompressed left-hand-side has 16 rows. CLA emulates this matrix-matrix multiplication via 16 vector-matrix multiplications. We observe AWARE performance comparable to multi-threaded ULA (sparse and dense) with improvements for Covtype, US Census, and US Census Enc, but a moderate slowdown for Airline and a significant slowdown for MNIST. LMM also shows a major performance difference when optimizing for memory versus optimizing for operations, which is especially noticeable in US Census Enc. In contrast, for datasets with smaller potential for co-coding like Airline, there is no difference. Figure 3.14(b) shows results on US Census Enc with varying number of rows in the left-hand-side. CLA performs similar to AWARE at a single row, but when the number of rows increases, CLA’s performance decreases to the same as ULA due to the lack of native matrix-matrix support. CLA is worse at utilizing more threads, while AWARE and ULA scale better. For ULA and AWARE-Mem, there is a change in parallelization strategies after 16 rows. In contrast, AWARE yields between half and one order of magnitude speedups for all #rows configurations.

Right Matrix Multiplication (RMM): In contrast to LMM, the right matrix multiplication creates outputs of overlapping column groups with a shallow copy of the index structures. Figure 3.15(a) shows the results for all datasets, where we observe AWARE speedups between 53x to 1,528x because of the deferred aggregation across column groups. Figure 3.15(b) then shows the scaling with increasing number of columns in the uncompressed right-hand-side. CLA shows equal performance to uncompressed in the single column case but scales worse than ULA, again due to the lack of native matrix-matrix multiplication. AWARE’s RMM exhibits better asymptotic behavior due to its dictionary-centric operations, yielding speedups >13,000x for 512 columns.

Transpose-Self Matrix Multiplication (TSMM): Figure 3.16(a) and Figure 3.16(b) show the results of TSMM operations as used for computing PCA, direct-solve linear regression,

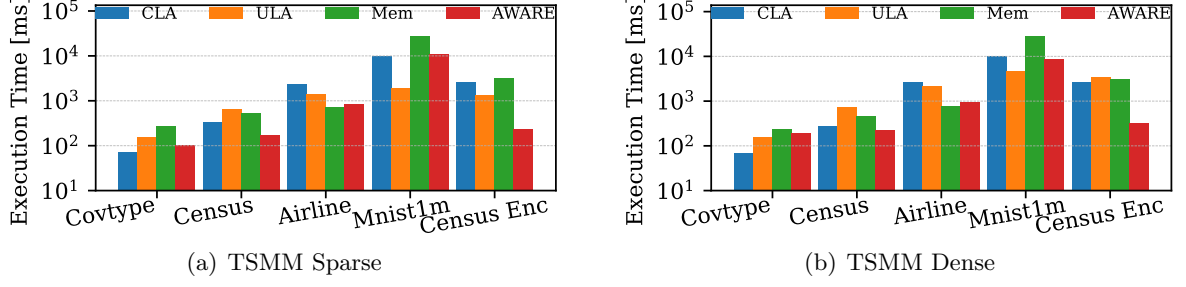


Figure 3.16: Operations Performance Transpose Self Matrix Multiplication.

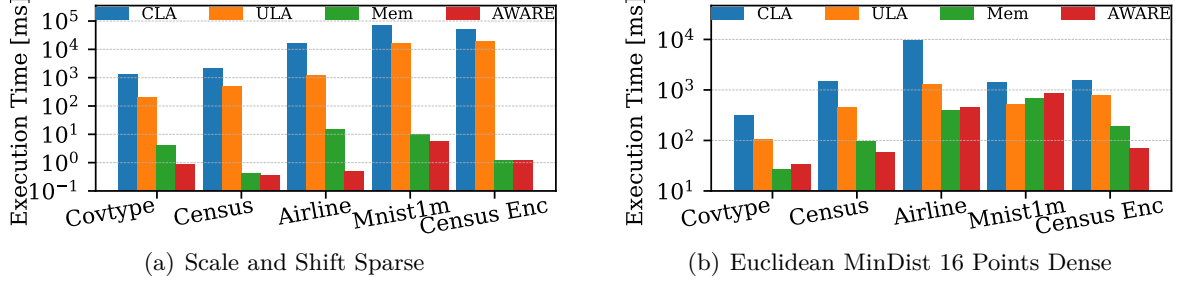


Figure 3.17: Operations Performance Sequence.

as well as covariance and correlation matrices. We observe speedups on all datasets except MNIST, where AWARE yields a substantial slowdown, especially for sparse inputs. The TSMM performance is largely dependent on the number of column groups, their number of distinct items, and thus, co-coding decisions. MNIST has a high number of columns, with high cardinality, and low correlation between columns.

3.4.4 Operation Sequences

Operation Sequences: As final micro benchmark use cases, we evaluate two sequences of operations. First, *scale and shift* in Figure 3.17(a) performs a shifting $\mathbf{Y} = \mathbf{X} - (\text{colSums}(\mathbf{X})/\text{nrow}(\mathbf{X}))$ and scaling $\mathbf{Z} = \mathbf{Y}/\sqrt{\text{colSums}(\mathbf{Y}^2)/(\text{nrow}(\mathbf{Y}) - 1)}$. This sequence is a common normalization step (standard-scaler) of the input data but has the negative side effect of densifying the input data. AWARE improves performance up to a best case of 15,399x. Second, we compute the minimum Euclidean distances via $\mathbf{D} = -2 \cdot (\mathbf{X} \times \mathbf{t}(\mathbf{C})) + \mathbf{t}(\text{rowSums}(\mathbf{C}^2))$, followed by $\mathbf{d} = \text{rowMins}(\mathbf{D})$ (which forces a decompression from overlapping state). Here, \mathbf{D} are the Euclidean distances of each row in \mathbf{X} to the centroids \mathbf{C} . This expressions is used, for instance, in K-Means clustering. AWARE shows performance up to 11.3x faster compared to ULA in all cases except MNIST.

Overlap: Leveraging the overlapping output from RMM without compaction shows significant improvements in Figure 3.15(a) and Figure 3.17. However, overlapping representations are most beneficial in chains of RMMs. Table 3.9 shows the end-to-end runtime for a sequence of 10 RMM of size $k = 512$, representative for processing 10 fully-connected layers of size 512 with no activation. CLA is slower than ULA in this scenario because it falls back to vector matrix compressed operations for the first multiplication. AWARE with no overlapping is slower because the first right multiplication decompresses, but it does show close to ULA performance. AWARE with overlapping column groups pushes the compressed index structures through the entire chain of RMMs, improving performance regardless of optimizing for memory or workload, with a slight advantage to workload.

3. Workload-aware CLA

Table 3.9: RMM Overlap Sequence (Data: US Census Enc).

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

Table 3.10: AWARE Workload TOPS (Data: US Census Enc).

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

3.4.5 Workload Cost Analysis

Computation Cost: Table 3.10 shows the AWARE workload analysis of different micro benchmarks on US Census Enc. This experiment shows the estimated Theoretical Operations (TOPS), calculated from the cost vectors and compression schemes. We compare the estimated TOPS for uncompressed operations (on the left) with AWARE’s estimated TOPS extracted from the sample and co-coding decisions, as well as the estimated TOPS after compression (on the right). We observe that the estimated TOPS from the sample is close to the actual TOPS, indicating good estimation accuracy and thus, meaningful costs. We also show the compression time (Comp) and the runtime (Time) for executing 100 repetitions of the given operation (Op 100×). There are some micro benchmarks that show disproportionate scaling of runtimes compared to TOPS. With small execution times, moderate discrepancies are expected because of various unaccounted overheads in both ULA and AWARE. For TSM and LMM, the differences are due to output allocation, memory bandwidth limitations, and index structure lookups. Although the runtime discrepancies are sub-par, we found that our TOPS estimation provides a good balance of simplicity and reflecting key differences relevant for compression. More sophisticated cost estimators are, however, interesting future work.

3.4.6 End-to-end Algorithms

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

Local Execution: Table 3.11 shows the results for local algorithm execution where data fits in memory. AWARE yields moderate but consistent improvements, or at worst (e.g. L2SVM, lmDS) comparable performance. Observing improvements on all algorithms most notably 19% for MLogReg, 47% for K-Means (iterative algorithms), and 29% for PCA (non-iterative algorithm) is remarkably because this includes online compression. Underlying reasons are fast compression that is easier to amortize and redundancy-exploiting operations. The algorithms L2SVM, lmCG, and lmDS all perform very close to ULA.

CLA Comparison: CLA is not included in Table 3.11 because it does not support scale&shift and therefore would not execute efficiently. For a fair comparison, we use the L2SVM algorithm from CLA [75] (with minor modifications, e.g. 60 iterations not 100) and compare different configurations of CLA (in SystemML) and AWARE in Table 3.12. Both systems read and parse both train and test datasets (in binary), increasing I/O compared to the other experiments. We observe that CLA compression is slower than AWARE optimizing for size or compute. CLA does not outperform ULA in SystemML in local settings because the compression is not amortized. In contrast, our ULA baseline is 1.9x faster, AWARE-Mem shows similar performance to CLA, and AWARE improves the relative training time (without compression and I/O) by 2x over CLA, and 2.3x over ULA, but SystemDS ULA is the fastest end-to-end. Since CLA mostly focuses on large distributed datasets, we further compare CLA and AWARE on a larger sparse dataset (256x, which only partly fits in memory of 11 nodes). Table 3.12 (right) shows that SystemML CLA yields a moderate speedup, but AWARE achieves another 2x over SystemML CLA. At this scale, AWARE optimizes for memory size and thus, the results AWARE and AWARE-Mem are similar.

Hybrid Execution: In between the local and distributed extremes, there are hybrid runtime plans, where the sparse input fits into the memory of the driver, but after scale&shift transformation, the transformed data does not fit in the driver and thus generates distributed operations. Table 3.13 show the results for replicated versions of US Census Enc (8x-32x). Runs using distributed operations are marked with D , and local compression times are included in parenthesis. These in-between scenarios are generally challenging because of evictions, efficient exchange between local and distributed runtimes, as well as decisions on when to prefer distributed operations. Most notable is this characteristic in ULA, which is sometimes faster for larger scales. The variance stems from various instructions execution fit in local memory, and therefore, more or less instructions are executed distributed. For instance, in PCA, the number of distributed instructions grows from 16x to 32x to, finally, 128x. Across all algorithms—except for a few instances—AWARE shows consistent improvements, especially if we focus on computation time (without the compression time).

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

Table 3.11: Workload-awareness on Local End-to-End Algorithms (Data: US Census Enc)

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

3. Workload-aware CLA

Table 3.12: L2SVM (without scale&shift, 60 iterations, Data: US Census Enc)

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

smaller `spark.memory.fraction`) in order to ensure stable results. For comparison, we also include previous results from our secondary cluster (128x*) using the same configuration as hybrid execution, which caused lost executors in some cases. Due to redundancy-exploitation and good compression ratios—even on tiles (see Table 3.8)—we observe large improvements of 2.8x for K-Means, 2x for MLogReg, 6.6x for lmCG, and 2.8x for L2SVM. In contrast, PCA and lmDS are non-iterative algorithms. On the secondary cluster, we observed up to 70x performance gains. The differences in relative improvements are due to faster networking and OS file system caching of evicted partitions due to more physical memory per node (256 GB versus 128 GB), which favors uncompressed (ULA) operations.

3.4.7 Hyper-parameter Tuning

Executing a single short ML training algorithm makes it hard to amortize the online compression. In practice, however, most time is spent in ML pipelines that involve outer loops for enumerating data augmentation pipelines, feature and model selection, hyper-parameter tuning, and model debugging. AWARE adapts to such more complex workloads by spending more time on compression (which is easily amortized) and optimizing for operation performance in the inner loops. Table 3.14 shows results for a basic GridSearch hyper-parameter tuning of the MLogReg algorithm. Even for a small number of $3 \cdot 3 \cdot 3 \cdot 5 = 90$ hyper-parameter configurations, AWARE improves the local runtime (including compression) by 3x, which is a promising result for wide practical applicability.

Table 3.13: End-to-End Algorithms Hybrid Execution [Seconds]
(Data: US Census Enc, D Marks Runs Including Distributed Operations).

	K-Means		PCA		MLogReg	
	ULA	AWARE	ULA	AWARE	ULA	AWARE
1x	51.6	(6) 27.1	12.7	(6) 9.4	32.0	(7) 26.0
8x	471.0	(26) 117.8	330.3	(26) 42.6	393.3	(29) 88.2
16x	D 484.3	(48) 183.9	D 76.3	(47) 67.5	D 570.3	(58) 144.2
32x	D 1,491.6	D 1,496.3	D 70.3	D 61.2	D 671.5	D 629.9
128x	D 17,819.0	D 6,298.0	D 137.0	D 140.3	D 3,502.9	D 1,710.6
*128x	D 33,039.0	D 11,616.0	D 269.0	D 259.0	D 50,998.0	D 8,599.6
	lmCG		lmDS		L2SVM	
	ULA	AWARE	ULA	AWARE	ULA	AWARE
1x	19.8	(6) 18.6	15.6	(6) 14.3	38.9	(6) 36.5
8x	366.2	(26) 60.6	334.4	(29) 51.5	405.2	(26) 115.4
16x	D 104.4	(44) 91.7	D 80.2	(50) 75.8	D 252.6	(56) 195.5
32x	D 264.6	D 105.3	D 91.5	D 70.8	D 433.2	D 479.4
128x	D 1,611.4	D 242.6	D 175.9	D 162.4	D 5,286.9	D 1,904.5
*128x	D 33,090.0	D 469.0	D 365.9	D 465.0	D 74,016.0	D 1,060.0

Table 3.14: GridSearch MLogReg (Data: US Census Enc).

ULA	AWARE-Mem	AWARE
274.3 sec	238.1 sec	92.6 sec

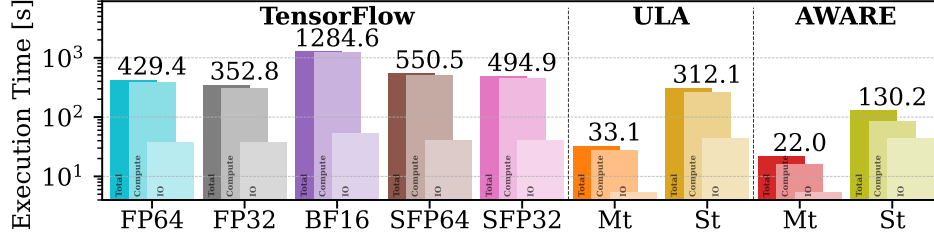


Figure 3.18: TensorFlow Comparison (lmCG, US Census Enc).

3.4.8 Comparison with Other Systems

While ULA is the most important baseline—within the same compiler and runtime—we also compare with TensorFlow (TF) version 2.12. We evaluated both TF and TF-AutoGraph [165], but report numbers for TF-AutoGraph, which gave 1-5 sec faster execution times on average. The workload is a simplified version of lmCG on US Census Enc, expressed via TF linear algebra operations. By default, we use 300 lmCG iterations (instead of 100 in Table 3.11).

Results: Figure 3.18 shows the results in log-scale, where each stack is I/O time, compute time, and total time (from front to back, as regular stacking is infeasible in log scale). On the left, we have TF with different value types. Changing from FP64 (double) to FP32 improves execution time by 21.7%, reducing to FP32 produces infinite sums, rendering the algorithm invalid. BF16 solves this issue by using different numbers of exponent and mantissa bits, but it is not well-supported on the CPU, resulting in a 3x slowdown compared to FP64. Using TF’s sparse representation worsens performance slightly at FP64 precision similarly at FP32. TF executes the core expression per iteration (of two matrix-vector multiplications) $\mathbf{X}^\top(\mathbf{X}\mathbf{v})$ single-threaded because it only uses multi-threaded matrix multiplications with two or more columns in the right-hand-side matrix. In contrast, our multi-threaded I/O and matrix-vector multiplications yield speedups of about 13x for ULA and 19.5x for AWARE. Forcing both single-threaded I/O and operations (St), ULA becomes 38% faster than TF. ULA (with data size of 1.3 GB) does not saturate the memory bandwidth for this sparse dataset, while AWARE fits the compressed matrix (49.7 MB) into the 128 MB L3 cache, yielding a 3.3x speedup over TF-FP64. To summarize, both ULA and AWARE show competitive performance with single-threaded and are faster with multi-threaded operations, indicating that AWARE’s improvements could carry over to other ML systems.

3.5 Summary

In this chapter, we introduced AWARE as a workload-aware, lossless matrix compression framework with new encoding schemes and compressed operations. Compared to previous lossless matrix compression, AWARE summarizes the workload characteristics of a linear algebra program and selects where and how to compress the inputs and intermediates to minimize total execution time.

Based on a variety of experiments, we draw two major conclusions. First, the broader spectrum of compression techniques (column groups, fast compression, overlapped representations) yields runtime improvements even when uncompressed operations fit in memory and can handle increasingly complex ML pipelines of data preparation, model training, and debugging. Second, the workload-aware compression planning nicely adapts the compressed representation for higher compression ratios when needed and otherwise prefers operation performance. Together, these characteristics yield a compression framework with robust performance and more general applicability.

The AWARE paper [23] proposed three directions for future work. Pushdown of compression into data preparation (e.g., feature transformations and data cleaning) [249], we present novel solutions to compressed preprocessing in [Chapter 4](#). Extensions for federated learning (e.g., extended asynchronous compression) [24, 25], for which we present preliminary results in [Chapter 5](#). Finally, combinations with lossy compression (e.g., bounded loss [121, 151]), that still to this day is promising future work [Chapter 6](#).

4

Compressed Transformations

The main parts of this chapter are under submission in BWARE.

Modern machine learning (ML) training comprises more than just selecting and fitting ML algorithms or neural network architectures as well as their hyper-parameters. Data-centric ML pipelines extend traditional ML pipelines of feature transformations and model training by additional pre-processing steps for data validation [206, 95], data cleaning [212], feature engineering [203], and data augmentation [196, 197, 232, 131] to construct high-quality datasets with good coverage of the target domain. These pre-processing techniques can substantially improve model accuracy [131, 212], the main target of most ML pipelines, but also other equally important measures like fairness [203, 219] and robustness [229].

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) [190] as well as data redundancy [23]. Besides natural data redundancy, such as the small cardinality of categorical features and column correlations, data-centric ML pipelines create additional redundancy. Examples are the construction of new data points or features, as well as systematic transformations such as the imputation of missing values by mean or mode and data cleaning by robust functional dependencies [66]. While beneficial for model quality, the iterative selection of such data-centric ML pipelines is expensive. Eliminating unnecessary redundancy through data reorganization is appealing because 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 [251]. Common layouts include compressed sparse rows (CSR), columns (CSC), or coordinate format (COO) [117, 175, 201, 215, 125]. 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) [75, 74], Tuple-oriented Compression (TOC) [147], Grammar-compressed Matrices (GCM) [77], and AWARE [23] (see Figure 4.1, top). AWARE creates compressed matrices (\mathbb{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 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. This conversion is a rich source of information about 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*, e.g., storage. Holistic support requires

(1) compressing the input 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 may differ, there is a need for morphing [101, 62, 58] compressed intermediate matrices into workload-optimized representations [23].

Contributions: In this chapter, we introduce BWARE (see Figure 4.1, bottom) as a holistic, lossless compression framework for data-centric ML pipelines. Our main contributions are:

- A lightweight frame compression scheme with dictionary encodings, enabling compressed feature transformations on heterogeneous data (Section 4.1).
- A morphing technique for workload-aware tuning of compressed representations (Section 4.2).
- Parallel and distributed I/O for compressed blocks without decompression (Section 4.3).
- An optimizing compiler injecting morphing instructions into LA programs (Section 4.4).
- An experimental evaluation that studies the impact of compressed I/O, feature engineering, feature transformations, and training in data-centric ML pipelines (Section 4.5).

4.1 Heterogeneous Frame Compression

This section describes BWARE’s frame compression, compressed feature transformations to matrices, and compressed feature engineering.

4.1.1 Compressed Frame Design

Uncompressed frames are tables stored in columnar arrays where each column can contain different value types. Figure 4.2 shows our Cframes using a dense dictionary compression (DDC) scheme per column. Each DDC column consists of a mapping array, $length = \#rows$, on the left and a dictionary array, $length = d_i$, on the right. d_i is d in column i . The map contains value positions in the dictionary.

Compressed Size: The compressed size depends on the number of distinct values d , value type, and number of rows. The mapping supports 0 or 1 Bit and 1-, 2-, 3-, or 4 Byte

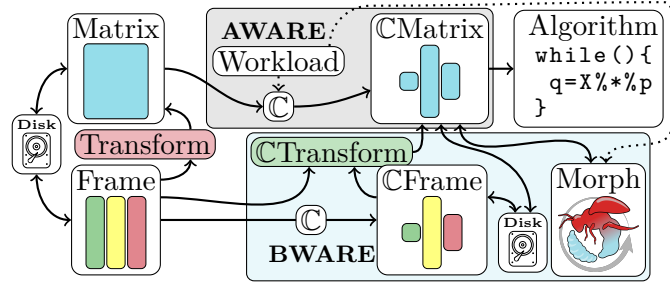


Figure 4.1: BWARE Framework Overview.

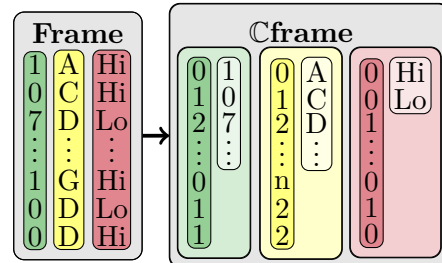


Figure 4.2: The Compressed Frame Format.

encodings per value $\#B$ (supporting up to 1, 2, 256, 64K, 16M, and 2G distinct values). If d_i is large, the dictionary allocation becomes too costly compared to the original value types, and we fall back to uncompressed arrays from the input frames columns. We compress boolean columns, which increases the size of the overall compressed format but can be leveraged in BWARE’s feature transformations and feature engineering.

Type Conversion: Type conversion improves allocations using 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, hex code, 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 columns because many feature transformations use unique dictionaries for individual columns, and different columns can contain different value types. Subsequent workload-aware morphing (Section 4.2) of the compressed format anyway tunes the final matrix compression with support of a wide variety of different encoding schemes. The proposed transformation techniques would also work with other dictionary-based compression techniques (e.g., RLE [3], SDC [23], and OLE [74]). Furthermore, we think it could be interesting to use it with GLA [77] and TOC [147] based column groups, all of which we leave for future work.

Compression: We fuse type detection, type conversion, and DDC compression. For each column: (1) We detect value types on a sample and try to apply the conversion while compressing. If the detected value type is invalid for the entire column, we revert to a full pass of the column to guarantee type detection. (2) To compress, we allocate a *mapping* that can encode as many unique IDs as rows. (3) Then, we iterate through all column rows and build a hashmap to encode unique column values with contiguously increasing IDs. All values IDs are stored in the *mapping*. (4) We pack the *mapping* into an improved format according to the d_i elements encountered to improve $\#B$. (5) We allocate a dictionary array, D , and fill it by looping through the hashmap’s key-value pairs $\langle k_i, v_i \rangle$ and assign $D[v_i] = k_i$. We do not compress the column if the hashmap grows too large compared to the $\#rows$ and detected value type. Even without DDC, type detection and conversion can still improve memory usage.

Parallelization: We naïvely parallelize over all input columns because the compressed frame format independently compresses columns. 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 `FP64` [141])—over row segments.

4.1.2 Compressed Feature Transformations

Transform-encode encodes a heterogeneous frame into a homogeneous matrix by applying dedicated feature transformations (built-in function `transformencode()`). The operation produces two outputs: A matrix and a metadata frame to apply the same transformations to other frames. We support the transformations shown in Table 4.1. Many other numeric transformations can be subsequently performed in linear algebra (e.g., normalization and scaling).

Lossless: We support two lossless transformations: *Pass* returns the same values as the input cast to double. *Pass* requires numeric inputs. *Recode* encodes input values into contiguous integers for each unique value encountered (similar to DDC encoding, just throwing away the dictionary).

Table 4.1: Transform Encode Types.

Name	OH	C-In&Out
Bin	✓	✓
Hash	✓	✓
Pass	✓	✓
Recode	✓	✓
Word Emb		✓

Lossy: Similarly, there are two lossy transformations: *Bin* short for Binning, constructs n 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. Finally, *hash* hashes each value and returns the hashed value modulo the maximum number of buckets to yield a bin ID.

One-hot Encoding: One-hot encoding (*OH*)—which is also called *dummy coding* (with subtle differences)—encodes contiguous integer values i into one-hot sparse vector representations with one set in cell i . Other transformations (which return integers) can use OH on top as specified in Table 4.1.

Word Embeddings: Encoding words into semantic-preserving numeric vectors is done via *word embeddings*, which is a sequence of *recoding*, *one-hot encoding*, and matrix multiplication with an embedding matrix. v denotes the size of each embedding vector. The matrix multiplication uses a selection matrix (details in Section 3.2.6.5), constructed via a contingency table on the recoded output.

Compression Sequences: Figure 4.3 shows different transformation sequences. The abbreviations F-CF is frame compression, and M-CM stands for matrix compression.

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 [23, 147, 74, 77]. However, the separate matrix compression has to extract statistics from the intermediate matrix again, many similar to the F-M transformation’s statistics.

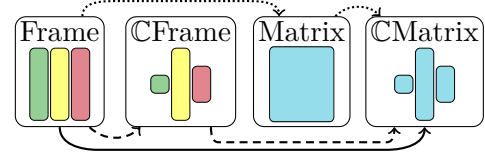


Figure 4.3: Transform Encode Sequences.

Frame to Compressed Matrix (F-CM): The compression feature transformations are:

- **Recoding:** Uses two passes: (1) construct a hashmap of unique values to continuous IDs, and (2) apply the assigned IDs. Finally, allocate a dictionary using the HashMap keys as values and values as offsets. **+Dummy:** use an identity matrix as a dictionary.
- **Pass-through:** Takes a sample if uncompressed and verifies compressibility. If the 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 the 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.
- **Bin:** Calculate the bin 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.

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 index structures (the map of DDC, but other structures for other compressions) allocated from 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 the transformation scale in the number of distinct values rather than the number of rows. The reusing approach is, however, only applicable if we use lossless transformations because lossy transformations have to reallocate or re-map their index structures.

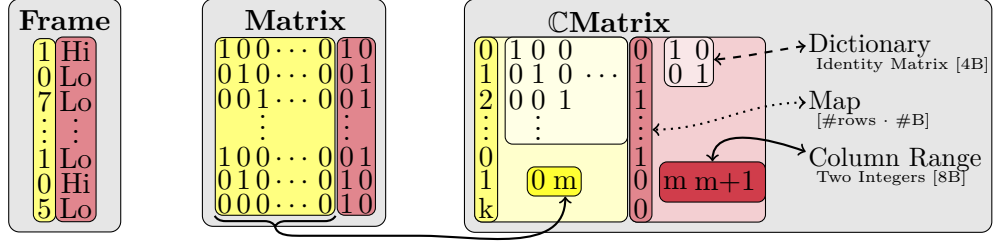


Figure 4.4: Recode and Dummy-code Two Columns.

Example: Figure 4.4 shows recoding and one-hot-encoding a frame of two columns. The uncompressed and compressed matrix results are shown in the middle and on the right. The transformation returns d_i matrix columns for each input column. The unique values are incrementally mapped to encoded values. The mapping from row indexes to dictionary entries is the value's recoded IDs. The dictionary is a virtual identity matrix (stored in a single integer). Each column group contains a column range with a start and end index. The mapping size depends on the number of rows and d_i . Assuming $|\mathbf{X}| = 1,000$ and $d_1 = 200$, the left mapping uses $1 B/\text{row}$ and the right column uses $1 b/\text{row}$. The Cmatrix then requires $1032 + 176 = 1208 B$ plus object/pointer overheads. If the input frame is compressed, like in Figure 4.2, the output can point to the input mapping of the compressed frame column. In such cases, the time complexity is $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$. Furthermore, in such cases with constant output, the allocation becomes constant with only $20 B$ of pointers.

Compressed Word Embedding:

Figure 4.5 shows how we perform a compressed word embedding for a single column input in $\mathcal{O}(1)$, 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.

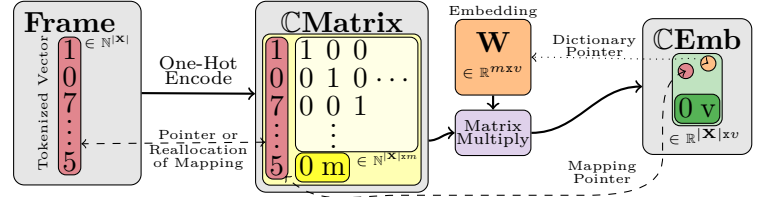


Figure 4.5: Compressed Linear Algebra Word Embedding.

Intermediate Sizes: Table 4.2 shows the sizes formulas of outputs. **F-M** is the uncompressed standard transformation, while **F-CM** and **CF-CM** produce compressed matrices. The first half of the table is without one-hot encoding. The one-hot **F-M** size assumes CSR output. Otherwise, dense representations require $8|X|d_i$. 'constant' means the compressed input index structure is used directly in the output.

Table 4.2: Transform-Encode Column Asymptotic Size.

	F-M	F-CM	CF-CM
Recode & Pass	$8 \mathbf{X} $	$\#B \mathbf{X} + 8d_i$	constant
Bin & Hash	$8 \mathbf{X} $	$\#B \mathbf{X} + 8\Delta$	$\#B \mathbf{X} + 8\Delta$
With One-Hot / Dummy-Coding			
Recode & Pass	$12 \mathbf{X} $	$\#B \mathbf{X} $	constant
Bin & Hash	$12 \mathbf{X} $	$\#B \mathbf{X} $	$\#B \mathbf{X} $
Word Embed	$(8v + 12) \mathbf{X} $	$\#B \mathbf{X} $	constant

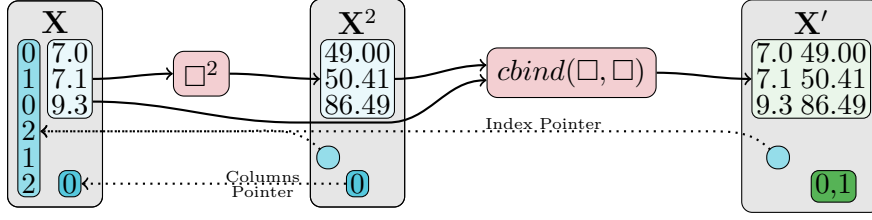


Figure 4.6: Extending Features in a Co-coded Column Group.

4.1.3 Compressed Feature Engineering

Feature engineering constructs or modifies features to improve the accuracy of ML models.

Modified Features: We define feature modifications as element-wise operations that change all instances of a feature equally. A typical operation is normalization. Normalization has multiple forms such as scale and shift or min-max scaling. Normalization is needed for well-behaved training of ML algorithms. Elementwise operations are supported in compressed space by prior work [77, 147, 23, 74]. In essence, we similarly perform most of the modifications with time complexity in the number of distinct items, $\mathcal{O}(d_i)$ for each column group.

Additional Features: One can add features by joining on keys or constructing new from existing features. An example is to expand \mathbf{X} with $\mathbf{X}' = \text{cbind}(\mathbf{X}, \mathbf{X}^2)$, where we concatenate \mathbf{X} and its squared representation. Such features allow simple linear models to take non-linearities into account, known as the kernel trick [15]. The append, however, requires allocating the full extended matrix. In contrast, BWARE combines new column groups with minimal additional allocations. Figure 4.6 shows the modification on a DDC encoding (but we support this operation for multiple compressed encoding types). The first step performs the scalar power operation. Scalar power is a dictionary-only operation, reducing the allocation to a new dictionary and maintaining pointers back to the original input mapping. When performing the column bind (*cbind*) operation, we detect that both indexes point to the same mapping, which indicates perfect correlation, and thus allows the direct combination into a co-coded column group. Similar exploitation strategies exist when subsets of columns are modified and appended. Furthermore, the compressed append can add multiple non-linearities at the same time $\mathbf{X}'' = \text{cbind}(\mathbf{X}, \mathbf{X}^2, \log(\mathbf{X}), \sqrt{\mathbf{X}})$.

Performance: $|\mathbf{X}|/d_i$ defines the potential speedup of compressed feature engineering of individual columns because the new features have a 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.

4.2 Morphing

Our novel morphing-based compression transforms uncompressed dense or sparse, as well as compressed matrices, into tuned compressed matrix representations. The morphing sequence is shown in Figure 4.7.

First, for uncompressed inputs, we extract column statistics from the input and group columns according to these statistics and workload. Second, in the case of a compressed matrix input, we directly reuse the statistics of pre-existing co-coded columns and skip unnecessary exploration. The result is a morphing/compression plan that contains a recipe for which columns to merge and what type of encoding to use.

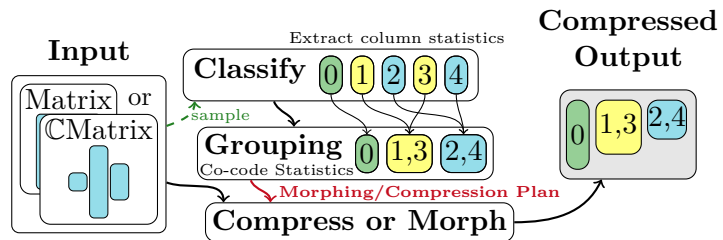


Figure 4.7: Morphing or Compression Sequence.

Algorithm 7 Morphing DDC Combine Algorithm.

Require: DDC_{Map} : I_1, I_2 , DDC_{Dict} : D_1, D_2 , DDC_{Cols} : C_1, C_2
 $\mathbf{M} \leftarrow \text{HASHMAP}$, $I_R \leftarrow \text{CONSTRUCTINDEX}(\text{LEN}(I_1))$
for i_1, i_2, i_R in I_1, I_2, I_R **do**
 $i_R \leftarrow \mathbf{M}.\text{PUTIFABSENT}(i_1 + i_2|D_1|, \mathbf{M}.\text{SIZE}())$
end for
 $D_R \leftarrow \text{CONSTRUCTDICTIONARY}(\text{SIZE}(\mathbf{M}), \text{LEN}(C_1) + \text{LEN}(C_2))$
for k, v in \mathbf{M} **do**
 $D_R[v] \leftarrow \text{COMBINE}(D_1.\text{GET}(k\%|D_1|), D_2.\text{GET}(k/|D_1|))$
end for
return $\text{DDCCOLGROUP}(I_R, D_R, \text{COMBINE}(C_1, C_2))$

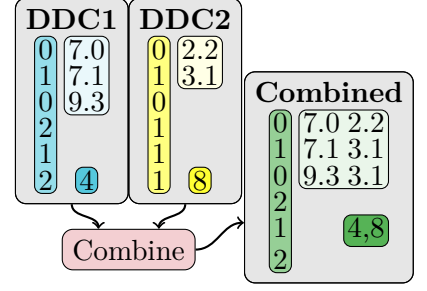


Figure 4.8: Combining
Two DDC Groups

Morphed Combining of Compressed Columns: To avoid decompression, we designed a co-coding algorithm that takes two encoded columns and returns a compressed co-coded column. We support combining various input column encodings. Algorithm 7 and Figure 4.8 show the combination of DDC column groups. The two column groups to combine are on the left, and the combined output is on the right. A naïve combination would produce the cartesian product of the dictionaries. Instead, our solution materializes only dictionary tuples that co-appear in the index structures. The cbind example from the previous section is a form of combining where the dictionary’s number of unique tuples does not grow. Each cell value in a naïve mapping can be calculated via $i_R = i_1 + i_2d_1$. Instead of populating the combined index with the naïve index values, we indirectly populate a hashmap to assign the combined index. Each combined dictionary tuple can then be looked up through the hashmap. The asymptotic runtime of creating such co-coded column groups is $\mathcal{O}(|\mathbf{X}| + d_R)$, where typically $d_R \ll |\mathbf{X}|$.

Morphing a Column Encoding: Combined column groups might not be in the correct encoding per the morphing plan. Therefore, the final step is to morph individual groups into other encoding types. Since most of our encodings are variations of dictionary encoding, the conversion is simple. We try to change encodings while reusing intermediates as much as possible. In practice, changing encodings typically only changes the index structure while keeping dictionaries.

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

4.3 Compressed I/O

Prior work on workload-aware lossless matrix compression used online compression after local or distributed reads, where—at least for local compression—the entire uncompressed input matrix needed to fit in memory. This approach restricts the size of compressible matrices and redundantly compresses the input matrix for every program execution. To address these limitations, we extended our BWARE compression framework to read and write compressed blocks and support continuous compression of streams (collections) of blocks.

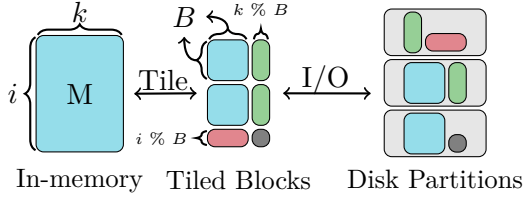


Figure 4.9: Uncompressed Tiled Format.

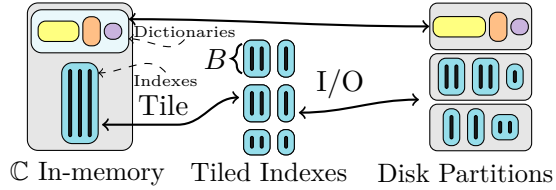


Figure 4.10: Compressed Tiled Format.

4.3.1 Compressed Data Layout

The on-disk format is a tiled format allowing distributed reads of collections of pairs of index and matrix blocks. Figure 4.9 shows the structure. The format excels in reading from local and distributed (HDFS) storage since multiple partitions can be read in parallel [65]. Reading and writing the matrix/frame formats have unique challenges to avoid decompression. When reading, we combine multiple, potentially differently compressed, blocks. When writing, we have to tile our index structures according to the block size selected. For both read and write, we support local and distributed operations.

Partitions & Tiles: Partitions hold multiple tiles and are written to individual files. The minimum number of tiles in each partition is determined by the partition sizes. We use minimum partition sizes of 128MB in HDFS (default block size) and 16KB in local (largest common disk block size). Partitions are allowed to grow larger than the minimum size.

Compressed Disk Format: The compressed format on disk can be seen in Figure 4.10. We split the compressed index structures and the dictionaries into separate partition files. For local writes, we write the dictionaries only once (into an optional dictionary file) and rely on the read logic to combine the indexes and dictionaries again (e.g., via Spark `broadcast` for distributed reads). However, blocks compressed independently in distributed operations save index structure and dictionary together similarly to the uncompressed formats. The difference in behavior avoids collecting the distributed compressed representations to deduplicate the dictionaries but reduces the compression ratio because of duplicate dictionaries.

4.3.2 Reading and Writing Compressed Data

Local Reading: When reading a compressed matrix to local memory, we combine all the blocks into one consolidated columnar compression scheme. Such a read happens when reading a compressed input from a disk but is also applicable when collecting outputs of Spark operations. The simplest case of combining is if all blocks in a column use the same compression scheme. In this case, only one of the dictionaries is collected, and each index structure can be combined directly. In other cases, it is more complicated and might lead to decompression or morphing of individual sub-blocks and re-compression. Morphing enables changing the compression scheme of sub-blocks into the consolidated scheme without decompression. However, selected group encodings have methods to combine directly with other types, such as `CONST`, `EMPTY` and `DDC`.

Distributed Reading: To read a compressed matrix in Spark, we construct a sequence of RDD (resilient distributed dataset) operations that lazily evaluate and materialize the compressed sub-blocks. If no separate dictionary file exists, all tiles should be self-contained (with index and dictionaries). In that case, we return a *PairRDD* of indexes and blocks. If there is a dictionary file, we first read the index structures into *PairRDDs* of matrix indexes and compressed index blocks and another of column index and dictionary blocks. We then join the dictionaries with the index structures to construct the combined compressed blocks. The result is a collection of self-contained compressed blocks. If the dictionaries grow above the tile size, then each distributed compressed block would be larger than uncompressed blocks. This does not happen in practice because before writing any compressed block (or sub-block), we check if an uncompressed dense or sparse version is smaller and use the smallest.

Algorithm 8 Serialized DDC Fused Update And Encode.

Require: *Matrix* : \mathbf{X} , Scheme: S ; $C \leftarrow S_{Cols}$, $M \leftarrow S_{Map}$, $D \leftarrow S_{Dict}$

```

 $m_s \leftarrow M.SIZE()$  ▷ HashMap size before updates
 $I_R \leftarrow CONSTRUCTINDEX(ROWS(\mathbf{X}))$  ▷ Allocate output index
for  $r$  in  $\mathbf{X}.ROWS()$  do
   $t \leftarrow EXTRACTTUPLE(r, \mathbf{X}, C)$  ▷ 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  $I_R[r] \leftarrow i$ 
  end if
end for
if  $m_s < M.SIZE()$  then  $D \leftarrow UPDATEDICT(M, D)$ 
end if
return DDCCOLGROUP( $I_R, D, C$ )

```

Update & Encode: To support large matrices or streaming use cases (e.g., continuous data collection), we can apply a compression plan on a stream of continuously arriving matrix blocks. The technique allows dynamic updates to a scheme to compress matrices without analysis, and each scheme can be applied in parallel. Algorithm 8 shows the DDC encoding variant (we support seven encodings in total). Given a compression scheme (determined from a sample), we first try to use a fused compression kernel that performs one pass over the input block. We use a one-pass algorithm because we are usually memory-bandwidth-bound. We start by allocating the output index structure and remembering the starting number of distinct values. The loop extracts the value tuples from the matrix depending on what columns are encoded. Then, we probe the map. If the tuple is not contained, we assign a new incremented ID otherwise, we use an existing ID. If we encounter many new distinct tuples, the index structure might be unable to encode them. In such cases, we abort and fall back to a two-pass algorithm that first updates and then encodes in two loops. If the map size is equal to the beginning, no new values are encountered, and we reuse the previous materialized dictionary. Otherwise, we update the dictionary. A benefit of this scheme is that *all* previously encoded blocks can use the latest dictionary for computations.

4.4 Compiler and Runtime Integration

Data-centric ML pipelines transform data through multiple stages from disk over pre-processing and augmentation to ML algorithms. Figure 4.11 shows a pipeline containing nested loops for finding the optimal pre-processing primitives. The stages comprise reading ①, a loop for different feature transformations specs t ②, a loop for augmentation strategies a ③, and the training loop of an algorithm ④, exemplified with a

```

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

```

Figure 4.11: Data-Centric ML Pipeline Pseudo-code

conjugate-gradient linear regression. There is potential to exploit redundancy via the previously presented techniques. However, adding compression to the stages would require hand-tuning the individual compound techniques based on the transformations, augmentations, and algorithms used. Instead, we propose an optimizing compiler and runtime that dynamically introduces the compression primitives in given linear algebra programs.

Compiler: We decide, at compile time, where to inject compression and morphing instructions. User-defined linear algebra programs, such as Figure 4.11, are first compiled into a hierarchy of statement blocks (for conditional control flow and function calls) containing

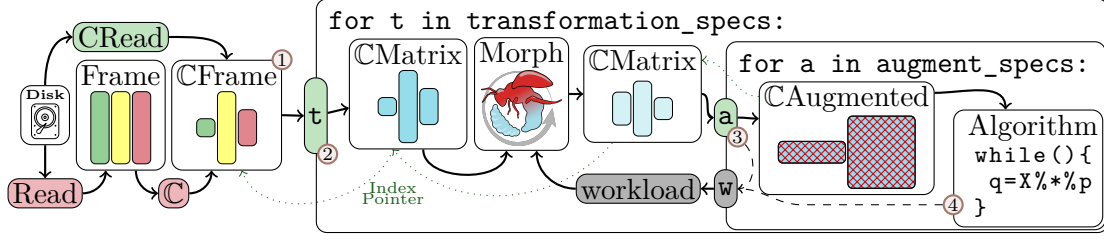


Figure 4.12: BWARE: Data-Centric ML Pipeline with Compiler Introduced Compression

directed acyclic graphs (DAG) of high-level operations (HOPs) per last-level statement block. Each HOP is compiled into 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 (e.g., `floor`) or comparisons (e.g., `<=`), which produce integer and boolean outputs. For each candidate HOP, we then construct a workload vector of affected, data-dependent operations summarizing the workload costs of the respective intermediate. If the workload summary indicates potential for improvement, the HOP is marked, appending a morphing LOP to its compiled sequence of LOPs.

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 data modification), adjusting the compression while still adhering to subsequent workload and data characteristics.

Example: Figure 4.12 shows the compiled execution plan for the script from Figure 4.11. In stage ①, 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 matrix, where index structures can be reused, and most 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 algorithms④.

4.5 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 evaluating polynomial feature engineering performance 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 a data-centric ML pipeline that iterates through multiple feature transformations. Finally, we compare the transformation performance to other systems.

4.5.1 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), 1 TB 3200 MHz DDR4 RAM (peak performance is 6.55 TFLOP/s), 16× SATA SSDs in RAID 0 for data, and an Intel Optane SSD DC P5800X for programs, scripts, and local evictions (in case live variables exceed the buffer pool size). Our software stack comprises Java 17.0.11, SystemDS 3.3.0*, Hadoop 3.3.6, and Spark 3.5.0.

Baselines: As baselines, we utilize SystemDS uncompressed I/O and operations (ULA), and compare to AWARE [23] for lossless matrix compression, as well as ML systems (TensorFlow 2.15 [158], SKLearn 1.4.1 [185, 45]), data management systems (Pandas 2.1 [224], Polars 0.20 [233]), and generic compression systems (ZStd 1.5.5-4 [76], Snappy 1.1.10.3 [90]).

Datasets We use multiple datasets, each having different data types and transformation requirements. Adult [26] (also called Census) is a sample from the person records in a 1990 U.S.

Table 4.3: Used Datasets and ML Tasks.

Dataset Name	# Rows	# Cols	Categorical	Numeric	Task
Adult [26]	32,561	15	9	6	Binary
CatInDat [198, 236]	900,000	24	16	8	Binary
Criteo Day 0 [118]	195,841,983	39	25	14	Binary
Crypto [228]	24,236,806	10	1	9	Regression
KDD98 [182]	96,367	481	135	334	Regression
Santander [191]	200,000	201	0	201	Binary
HomeCredit [166]	307,511	121	16	105	Binary
Salaries [19]	397	6	3	3	Binary
AMiner V16 [222]	$\approx 4,000,000$	1,000	1,000	0	Word Embed

census. CatInDat [198, 236] (Cat) is a synthetic dataset from Kaggle that contains categorical features for predicting cat ownership, we combined the two competition datasets. Criteo [118] is a dataset of millions of display advertisements for predicting which ads were clicked. Criteo10M is the first 10 million rows from Criteo. Crypto [228] is a Google competition dataset for time series forecasting of crypto-currencies. KDD98 [182] is a knowledge discovery competition dataset from 1998. Salaries [19] is a small dataset containing the salaries of professors in a U.S. college. Santander [191] is another Kaggle competition to predict customer transactions. HomeCredit [166] predicts how likely each applicant is to repay a loan. AMiner V16 [222] contains ≈ 4 million abstracts from various conferences. We preprocessed AMiner by removing non-English abstracts, equations, and symbols. The datasets are summarized in Table 4.3. Categorical and numeric is the number of respective feature types. The tasks are split into regression, binary classification, and word embedding tasks.

4.5.2 Frame Compression and I/O

We first evaluate the sizes of compressed frames and I/O performance in Figure 4.13.

In-memory Size: Figure 4.13’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 values 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 improves it 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. The results show that BWARE can keep larger frames using less memory and guarantees (except for boolean data) less than or equal sizes to *Detect*. Interestingly, BWARE reduces allocation even in the tiny Salaries dataset.

On-disk Size: The second row in Figure 4.13 shows the size of the datasets on disk. The first column in each figure shows the original allocations (CSV files). The second column contains 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. The general conclusion is that BWARE performs almost equally to other compression frameworks for individual block compression, and we can recursively stack compression schemes for improvements.

Write/Read Time: The third row in Figure 4.13 shows the execution time to write the different formats. The writing time includes schema detection, schema application, and compression, if applicable. We see a moderate overhead for compression, but it is comparable to other compressors. The last row in Figure 4.13 shows the reading performance from CSV

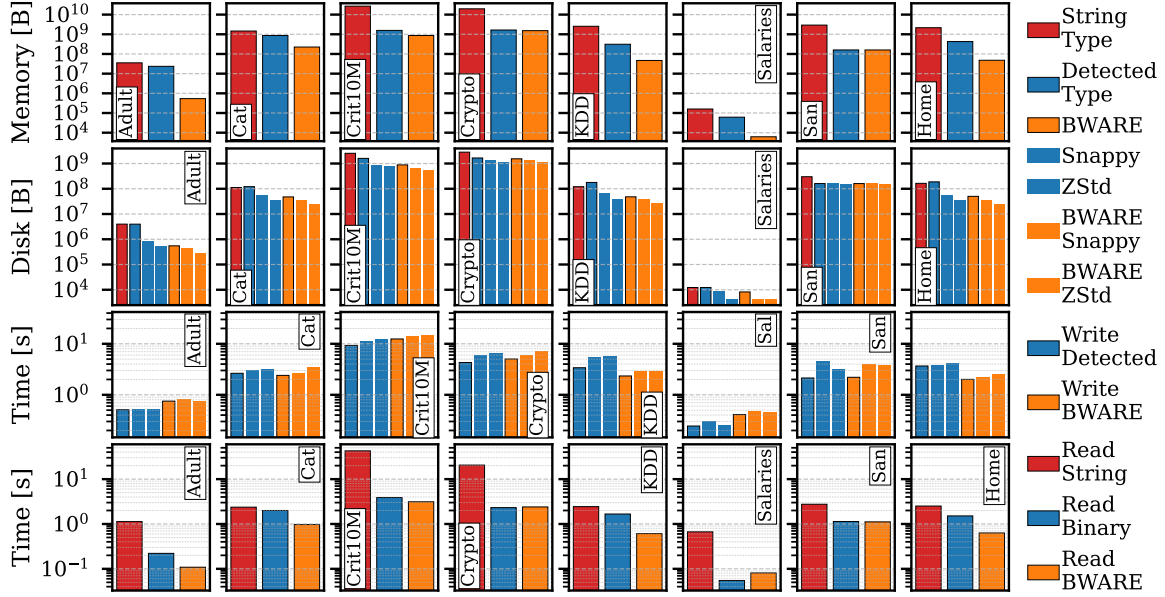


Figure 4.13: Frame Compression.

files as well as uncompressed and compressed binary files. We see 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.

4.5.3 Compressed Feature Transformations

Next, we evaluate feature transformations. We start with lossless, followed by lossy encodings.

Lossless Encoding: We one-hot-encode all categorical and pass-through numeric features in a lossless encoding. With this scheme, we copy over the values of Crypto and Salaries because all values are numeric, while most columns in Cat and Criteo are one-hot-encoded. Figure 4.14 shows the performance. The rows include (1) the matrix’s last size in memory, (2) the saved size on disk, and (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 allocation is close but consistently smaller than BWARE. However, BWARE reuses the index structure from the compressed frame arrays in cases with a 1-to-1 mapping from the frame column’s values. We further see that BWARE’s on-disk representation is slightly worse than AWARE (which we extended to use BWARE’s I/O operations).

Lossless Time: *String* encodes string types as input, while Binary (F-M) encodes using the detected types. AWARE (F-M-CM) encodes detected types followed by compression from scratch, and BWARE (F-CM) uses compressed transform encoding. BWARE is faster than the other solutions, with exceptions in the incompressible cases of Crypto and Santander. AWARE’s compression of Criteo shows what happens when the rediscovering of column

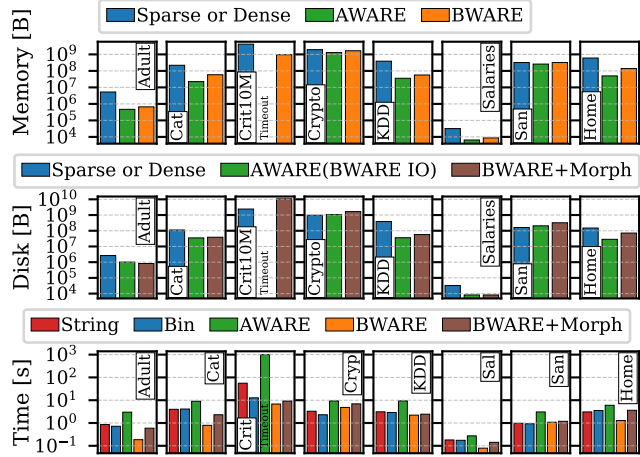


Figure 4.14: Lossless Transform-Encode.

correlations dominates. In essence, 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 and each co-coding candidate taking time to verify, we timeout the runs at 1000 seconds.

Lossy Encoding: Figure 4.15 shows results with different Δ (#bins) for numeric ranges or categorical hash buckets on the x-axis. 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 allocation. We observe that the morphed allocation is close to AWARE’s saved format. The results show that BWARE is faster across all datasets while being on par for Santander. BWARE yields improvements of 2x to 20x over AWARE and 1x to 5x over the baseline SystemDS.

Encoding Time Breakdown: All experiments, so far, used uncompressed frame inputs. If the input frame’s columns are compressed, the asymptotic runtime changes to constant for some transformations. Figure 4.16 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 are uncompressed. The constant encoding time can be seen in the plateau of the first 50% of the columns in the plot. The constant groups take $\sim 40ms$, except for 10m where it consistently is $\sim 100ms$. 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.

4.5.4 Compressed Word Embeddings

Figure 4.17 shows the performance of encoding already tokenized abstracts from DBLP [222], capped at a maximum of 1,000 tokens. All plots show the total execution time of 10 repetitions of performing word embedding with word2vec [161] embeddings trained on Wikipedia. The first row represents word embedding only, while the second row adds a fully connected

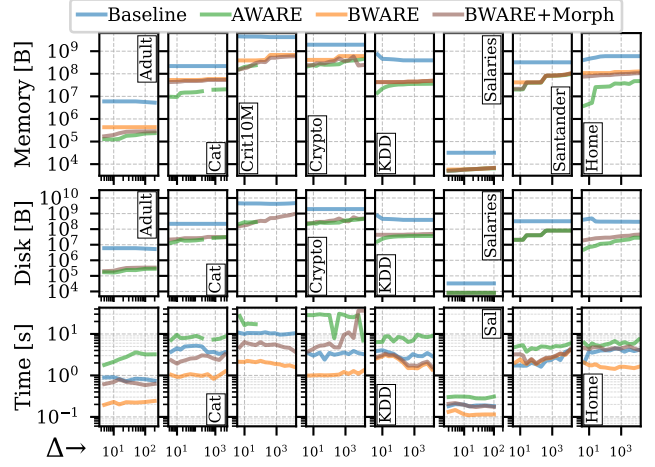


Figure 4.15: Lossy Transform-Encode Size and Time.

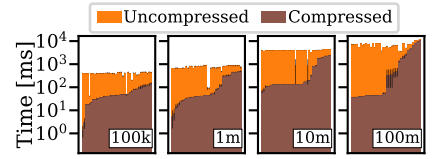


Figure 4.16: Compressed Input Frames.

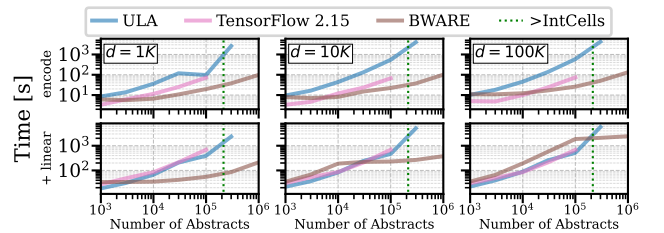


Figure 4.17: Compressed Word Embedding

neural network layer with ReLU activation

on the embedded outputs. The columns show increasing numbers of unique tokens (d) allowed, starting at $d = 1k$ and increasing to $100k$. The x-axis on all plots is the number of abstracts encoded, while the y-axis is execution time. We observe that ULA is slower at embedding but as fast as TensorFlow once the neural 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 the number of abstracts is lower than d . However, once the number of abstracts is larger than d , BWARE asymptotically and empirically outperforms all the other implementations.

4.5.5 ML Algorithm Performance

To quantify BWARE’s effect, we evaluate a conjugate gradient linear model training with different lossless, lossy, and feature engineering pipelines showcasing individual effects BWARE has on the combined performance of end-to-end ML pipelines.

Lossless: Figure 4.18 shows the performance of training a linear regression conjugate gradient method with L2 regularization. The max iterations are set to $\min(\#col, 1000)$. The algorithms are sparse-safe, allowing the baseline to take full advantage of sparse linear algebra. We observe a small slowdown using BWARE in some of the cases 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 are expected to have a small overhead in analysis and compression, but we observe that both the incompressible cases of Crypto and Santander keep the same execution time because the compressed transformations fall back to uncompressed representations. We do not show the accuracy of the models because the results of both solutions are the same. However, as an example, the method scores 78.9 AUC for Cat on Kaggle with such a simple model, while the top score is 80% [198].

Lossy: Figure 4.19 shows the results when controlling the number of distinct values through lossy transformations. The Crypto dataset is almost purely numeric and a dense dataset. When varying Δ , we observe baseline performance

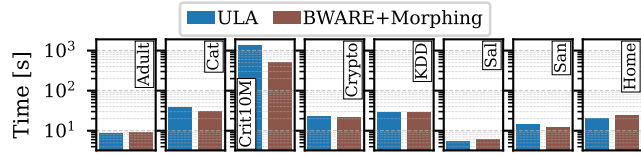


Figure 4.18: LM Conjugate Gradient Baseline.

similar to the lossless solution. The performance is expected since there is no benefit from reducing the number of unique values in uncompressed linear algebra. BWARE is able to exploit the reduced number of unique values, with a slight increase in run times when Δ increases. There are cases that do not benefit, such as KDD, where only extreme 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 $\Delta = 800$. The model is 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.

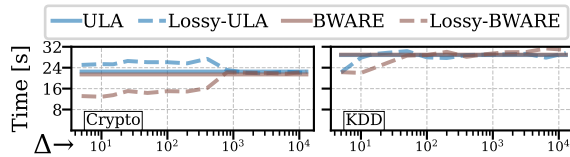


Figure 4.19: LM Conjugate Gradient Loss w/ Increasing Δ .

Scaling: Figure 4.20 shows the scalability of BWARE in terms of performance on larger subsets of the Criteo dataset. We observe a starting 2x performance improvement in Figure 4.18 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 31,792 to 2,880 sec. With lossy encodings of Criteo, both ULA and BWARE show similarly improved performance, with an increasing gap for more rows.

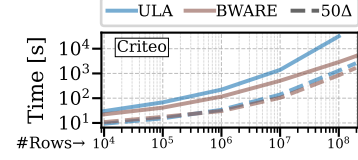


Figure 4.20: LM Training Scaling Rows.

Polynomial Regression: Figure 4.21 shows the polynomial regression (leveraging the kernel trick) results. These results indicate that BWARE facilitates polynomial feature engineering with very moderate costs and sometimes improvements. The best case is Crypto, where the polynomial features do not affect the execution time when combined with lossy feature transformations. The other datasets do not have significant performance improvements. BWARE performs poorly with lossless feature transformations in the Home dataset at low polynomials because the dataset contains a few incompressible columns (see Figure 1.3), which do not amortize with lossless feature transformations. However, once these columns are transformed in a lossy manner, the performance is good.

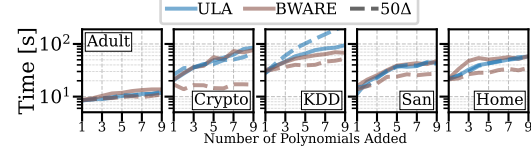


Figure 4.21: Kernel Trick Polynomial Regression.

Other Algorithms: AWARE already studied the impact of compression on multiple linear-algebra-based ML algorithms, which we inherit for BWARE. Figure 4.22 shows the performance of a few other 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.

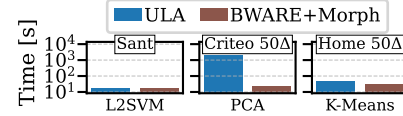


Figure 4.22: Various ML Algorithms.

Table 4.4: Pipeline LM: 8 Transform Encode & 8 Polynomials.

Dataset	Measure	ULA	AWARE	BWARE
KDD98	Execution 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 Consumption [173]	338kJ	185kJ	92kJ
Home	Execution Time	431s	266s	160s
Adult	Execution Time	19.9s	22.1s	16.6s
Santander	Execution Time	489s	376s	374s
Cat	Execution Time	467s	170s	63s

4.5.6 Data-centric ML Pipeline

Table 4.4 shows the execution time and characteristics of a full, end-to-end, data-centric ML pipeline similar to Figure 4.12. The table contains results from a pipeline performing a grid search of hyper-parameters with eight different Δ ranging from 5 to 480 and eight polynomials from 1 to 8. The pipeline uses two outer loops: the first performing **transformencode**, and the second polynomial feature construction. The top half of the table contains 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 feature transformations, and AWARE redundantly rediscovers correlated columns. BWARE’s handling of polynomial features is also used by AWARE in this experiment. AWARE and BWARE compression 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 [109]. 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 for a small overhead on Adult where the compression overhead cannot be amortized.

4.5.7 Comparisons with Other Systems

Finally, Figure 4.23 compares the performance of **transformencode** on Criteo with other systems. The left plot (E2E) is end-to-end times with CSV parsing, and the right (TE) is 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 [189]

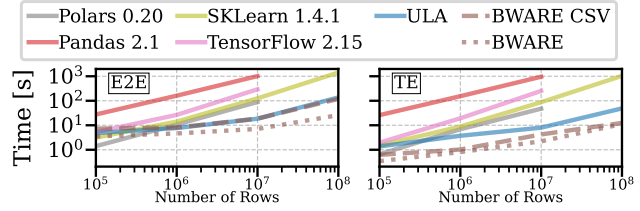


Figure 4.23: System Performance Comparison.

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 times 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 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.

4.6 Summary

This chapter introduced BWARE, a holistic, lossless compression framework for data-centric ML pipelines, integrated into SystemDS [31, 33]. In this context, we push compression through storage, I/O, feature transformations and feature engineering. We draw two main conclusions. First, this compression strategy can 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 data locality. Interesting future work includes support for more feature transformations (e.g., image data augmentation) and specialized, heterogeneous hardware accelerators.

5

Asynchronous Compression in Federated Learning

Parts of this chapter are from the ExDRA paper and a demo paper [24, 25].

Federated learning allows the training of machine learning (ML) models without collecting raw data in a centralized place. However, with this broad definition, there are many design decisions associated with building a federated system. The central question in federated learning is what can be shared from the federated sites. Data privacy requirements, data ownership, and other constraints define what is allowed to be done with the data. These constraints can hinder central data consolidation, which is typical for training ML models. To maintain even stricter constraints privacy-preserving ML can be used via multiple techniques.

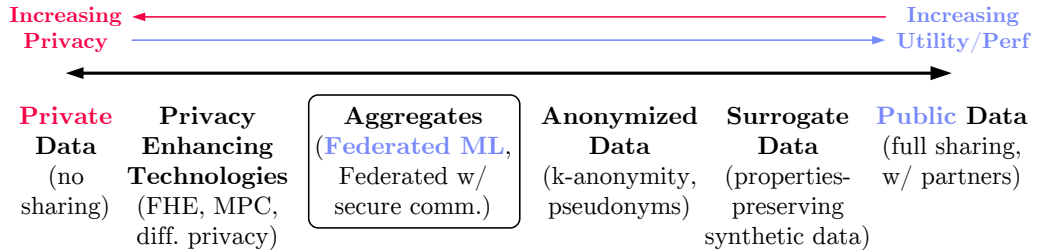


Figure 5.1: Spectrum of Data Sharing

Data Sharing Spectrum: Figure 5.1 shows a spectrum of different approaches to general data sharing. On the extreme left, no data is shared, meaning we would be unable to train on the data, and therefore, there is no utility of the data. Fully homomorphic encryption (FHE) [9, 83, 87], multi-party computation (MPC) [164], and differential privacy [119] provide the strongest privacy guarantees while allowing training on the data with limited utility. While maintaining strong privacy guarantees, the limited utility and high overhead make it impractical in many cases for training in larger ML pipelines. On the right, the data is freely available for sharing, marking the default training case of most ML systems. A more private alternative to full sharing is surrogate data, which synthesizes data based on real datasets and anonymized datasets. Both these alternatives improve privacy at the cost of a bit of utility.

Federated Learning: Federated Learning [122, 40] overcomes the sharing problem by performing decentralized computations that push operations directly to their data sources. A common architecture for performing federated learning is a data-parallel parameter server [64, 213, 148], adopted for instance in TensorFlow [5] and PyTorch [149].

Parameter Server: In the general parameter server architecture, we initialize a central coordinator with a model. The federated sites act as workers that pull the central model and perform model updates via forward and backward passes through a neural network. The passes produce gradients that are sent back to the central coordinator. The coordinator accumulates all the gradients from all the federated sites and updates the global model. Once the model is updated, the loop starts over with federated sites pulling the new model.

The Federated Data Sharing: If done right, the data sharing of federated learning only reveals data distributions but not the underlying raw data. Therefore, federated learning adds another point to the spectrum of data sharing, where only aggregates, for instance, in the form of gradient updates, are shared. Federated learning is more private than anonymized data but less than FHE, MPC, and differential privacy. The intermediate positioning of federated data sharing enables high levels of privacy together with a high degree of utility [24].

Data Science Workflows: Typical data science projects deal with open-ended questions to create business value by finding interesting patterns and constructing various models on data [31]. The process is exploratory, where analysis and results guide the refinement of pipelines [67, 248], and due to the exploratory nature, little investment is made into systematic data acquisition, integration and pre-processing [208]. Furthermore, data typically is stored in raw (potentially distributed) formats where repeated downloads, reads, and similar preprocessing and storage overhead can be reduced by leveraging standing federated processes and decentralized computation.

Contributions: This chapter describes the architecture of the federated ML backend integrated into Apache SystemDS [24]. Our detailed contributions include:

- *System Architecture:* Section 5.1, we describe the federated system architecture of SystemDS and its components for model and pipeline management, federated ML and data preparation.
- *Federated ML Runtime:* Section 5.2 describes the federated runtime of SystemDS, federated data organization and compression (Section 5.3), as well as federated linear algebra, parameter servers and data transformations.
- *Federated Runtime Experiments:* Section 5.4 shows experimental results of various local and federated ML algorithms and pipelines. Including standing workers’ dynamic workload-aware compression of data.

5.1 Federated Primitives

Before defining the federated system architecture of SystemDS, we cover two general federated properties relevant to the federated compression of intermediates. First, the semantics of federated data, followed by techniques for maintaining privacy in the federated setting.

5.1.1 Federated Data

In a federated setting, the raw data remains at the individual federated sites. Operations are pushed to the sites, and results are collected from individuals or chains of pushed operations. For data analysis and model building, multiple back-and-forth transfers of intermediate results can be performed. As a primary concept, federated data, a frame or a matrix, can be defined as arbitrarily (most likely non-overlapping) disjoint regions of data. However, two cases are predominant.

Row-partitioned: Row partitioned data—or horizontal federated learning [40, 252]—comprises data partitions of full rows, where all federated sites contain different row segments. Conceptually, row-partitioned data contains all categories of features. In other words, there is no site that has any extra features compared to others. In the row-partitioned case, labels for

the rows can be located on the federated sites or uniquely assigned based on the federated sites from the coordinator.

Column-Partitioned: Vertically partitioned federated learning [252, 247], also called column-partitioned federated data, is less common and refers to distributed subsets of features. The distributed features can be partially overlapping. Examples include site-specific sensors or distributed heterogeneous data collection. Combining the different features together gives a richer feature set for training ML pipelines.

5.1.2 Federated Privacy

To maintain the privacy of the data, federated learning can employ various strategies. A key property to maintain is that any shared information does not allow reconstruction of the private data of federated sites. To achieve this level of privacy, there are multiple techniques that can be employed.

Aggregates: First, most ML models can be fitted to federated data while restricting the communication to transfer only aggregates (e.g., gradients). However, to maintain privacy, any aggregate has to contain a sufficient number of elements, and aggregation operators, via, for instance, matrix multiplication, have to be sufficiently complex to not reveal underlying data.

A Limitation: The federated backend in SystemDS does currently not verify the complexity of the broadcasted variables to ensure the privacy of returned results. An example of an adversarial input is a weight matrix with a single non-zero such an operation essentially selects a tuple to be collected by the centralized coordinator. While this example is simple to find, it is complex to define the requirements of all possible adversarial intermediates that would break privacy. Therefore, evaluating each weight matrix transferred to calculate gradients would be expensive.

Encrypted Communication: An additional and orthogonal privacy enhancing method is encrypted communication that—via standard encryption techniques—ensures that intermediates are only shared with trusted parties. This is necessary to ensure safe communications between the coordinator or federated sites in case the connection is unsafe and not otherwise sufficiently covered.

5.2 Federated Runtime

We build a federated backend into SystemDS that is able to compile DML-based scripts into hybrid execution plans containing local, distributed, and federated operations. Federated instructions execute linear algebra primitives on federated data sources.

5.2.1 Federated Backend

To use the federated capabilities of SystemDS, we initialize federated data objects that inform the compiler to subsequently use federated instructions for processing. Figure 5.2 shows this setup and integration of the federated backend.

Federated Data: We support federated matrices and frames. A centralized controller acts as the coordinator and holds metadata—in the form of a federation map—of federated data. The map stores the data type, value type, dimension, and sparsity, as well as non-overlapping data ranges and their locations (host, port, and path). For example, in Figure 5.2 we have a federated matrix of $100K \times 70$ matrix \mathbf{X} , with distributed row partitions $[1 : 40K]$, $[40K : 80K]$, and $[80K : 100K]$ on node1, node2, and node3, respectively. If a local operation at the coordinator tries to pin \mathbf{X} into memory, the federated data is transparently transferred—unless it violates privacy constraints—and collected into a local matrix on the controller.

Federated Workers: Similar to the coordinator, the federated workers are also control programs but started as worker processes that act like permanently running servers at the federated sites. A worker listens on an input queue for incoming RPC requests (called

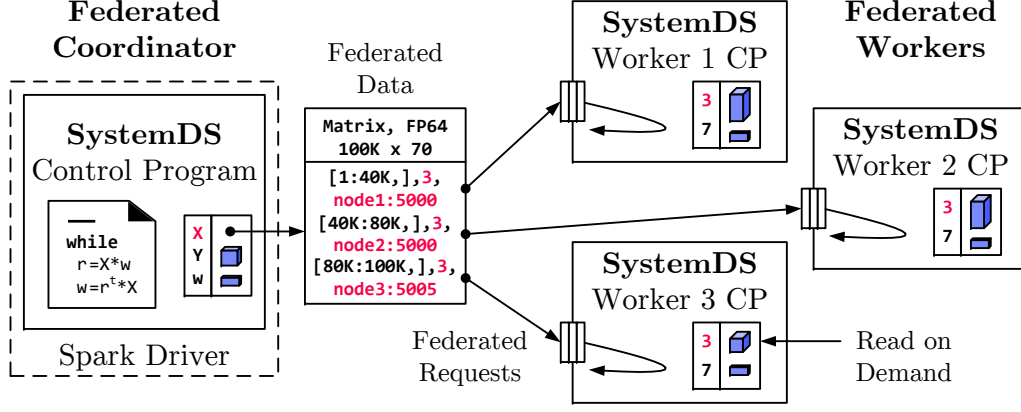


Figure 5.2: Federated Runtime Backend

federated requests), executes these requests, maintains a local symbol table, checks privacy constraints, and returns an RPC response. This design of worker CP programs provides very good flexibility and reuses the I/O subsystem, buffer pool mechanisms, as well as local and distributed operations. For example, it even allows data center federation [235], where a single federated operation triggers distributed operations in a Spark [255] or Flink [16] cluster at the federated site.

Initialization: A user can initiate a federated matrix or frame from federated configurations, files containing a list of federated sites and shapes. The domain-specific language (DSL) in SystemDS DML can instantiate a federated matrix with:

```
X = federated(addresses = list(a, b),
              ranges = list(list(0,0), list(200, 10),
                           list(200, 0), list(400, 10)))
```

This matrix consists of two federated sites with row-partitioned data *a* and *b*, each site containing 200 rows and 10 columns of data. It can also be allocated in the Python API:

```
from systemds.context import SystemDSContext
with SystemDSContext() as sds:
    X = sds.federated([a,b], [[0,0], [200,10]],
                     ([200,0], [400,10]))
```

The federated *X* matrix can subsequently be used in any linear algebra program, and SystemDS automatically compiles federated instructions where applicable. Instead of providing a labor-intensive implementation of individual federated ML algorithms, we aim to simplify the development and execution of federated scripts.

Federated Requests: The coordinator communicates with workers through federated requests, using Netty (also used in Spark) as a network I/O framework for RPCs and data transfers. To simplify the implementation of federated operations, we restricted the federation protocol to only six generic request types simplifying the design:

- **READ(ID, fname):** Creates a data object from a filename, reads it, and adds it by ID to the symbol table.
- **PUT(ID, data):** Receives a transferred data object, and adds it by given ID to the symbol table.
- **GET(ID):** Obtains a data object from the federated site’s symbol table, and returns it to the coordinator.
- **EXEC_INST(inst):** Executes an instruction, which accesses inputs and outputs by IDs in the symbol table.

- **EXEC_UDF(udf)**: Receives a serialized, user-defined function (UDF) object, executes this UDF over requested inputs by ID, may add outputs to the symbol table, and returns a custom object to the coordinator.
- **CLEAR**: Cleans up execution contexts and variables.

For efficiency, the coordinator sends RPCs to all workers in parallel, and a single RPC can contain a sequence of requests and returns a single response. The simplicity of these request types has two profound advantages. First, we can reuse existing instructions for composing federated operations. Second, this design allows for federation hierarchies. If the worker-local data is federated data, a worker can also act as a coordinator of a subgroup of workers.

5.2.2 Federated Linear Algebra

For broad applicability in various ML algorithms and data science lifecycle tasks, our federated runtime supports both federated linear algebra and federated parameter servers. Federated linear algebra utilizes similar strategies as distributed, data-parallel operations but retains the raw federated data at its federated site. This requirement creates additional challenges and needs compiler support for finding valid yet efficient runtime plans if operations do not directly apply.

Basic Linear Algebra: During compilation and runtime, we check if any inputs are federated data, and dispatch this call to supported federated instructions. Similar to RDD transformations and actions [255], these federated instructions then utilize federated requests—and related high-level primitives for broadcasting and aggregation—to compute the operations over federated data. If no aggregation is needed, the output is itself federated data.

Example: Federated Matrix Multiplication: *Assume a matrix-vector (or matrix-matrix with small right-hand-side) multiplication $\mathbf{X}\mathbf{v}$ and vector-matrix multiplication $\mathbf{v}^\top \mathbf{X}$ with $\text{nrow}(\mathbf{X}) \gg \text{ncol}(\mathbf{X})$ and \mathbf{X} being composed of federated row partitions. For matrix-vector, we broadcast \mathbf{v} via PUT, execute a local matrix-vector multiplication per partition via EXEC_INST, which yields a new federated vector with related federation map (logical rbind here), and finally execute an optional rmvar instruction via EXEC_INST to clean up the broadcast \mathbf{v} . In contrast, for a vector-matrix, we perform a sliced broadcast of \mathbf{v} (vector parts according to row ranges), execute a local vector-matrix multiplication per partition via EXEC_INST, obtain the results via GET, do a final aggregation via element-wise vector addition at the coordinator, and again, clean up all intermediates.*

Supported Operations: So far, we support—as summarized in Table 5.1—federated matrix multiplication, unary aggregates, unary element-wise operations, binary matrix-matrix, matrix-vector, and matrix-scalar operations, ternary, quaternary, and parameterized builtin operations, and various reorganizations. These operations further support both row- and column-partitioned federated data via specialized implementations. Most of the binary operations (e.g., matmult, element-wise) support a single federated input and consolidate a second federated input (e.g., aggregated intermediates) in the coordinator. However, whenever two federated inputs are co-partitioned (e.g., because one originated from the other), we directly execute federated operations on them as well.

Higher-level Primitives: SystemDS follows the premise that many data science lifecycle tasks—like data validation, data cleaning, feature and model selection, and model debugging—are themselves based on machine learning and numerical computation [31]. These higher-level primitives are hierarchically composed from built-in functions that rely on linear algebra and thus, are directly supported on federated data as well. In case a binary or ternary operation is not supported over multiple federated matrices, some of them are consolidated in the coordinator, or a privacy exception is thrown if this consolidation would reveal private raw

Table 5.1: Example Federated Instructions.
(Since ExDRA [24], we support many more operations)

Operation Type	Examples
Matmult	mm, tsmm, mmchain
Aggregates	sum, min, max, sd, var, mean rowSums, ..., rowMeans, colSums, ..., colMeans
Unary	abs, cos, exp, floor, isNA, log, !, round, sin, sign, softmax, sqrt, tan, sigmoid,
Binary	&, cov, cm /, =, >, >=, %/%, <, <=, log, max, min, max, -, %%, *, !=, , +, ^, xor
Ternary	ctable, ifelse, +*, -*
Quaternary	wcemm, wdivmm, wsigmoid, wsloss
Transform/Reorg	tfencode, tfapply, tfdecode, rbind, cbind, t (transpose), removeEmpty replace, reshape, X[:, :] (matrix indexing)

data. For this reason, we are working toward better compiler support that proactively considers privacy constraints and generates efficient runtime plans that adhere to these constraints.

Example: Federated K-Means: *Starting bottom-up, individual ML algorithms are good examples of such higher-level primitives. For instance, consider the inner loop (after initialization and inside a loop for multiple runs) of K-Means clustering, where \mathbf{X} is a federated, row-partitioned matrix, and \mathbf{C} are the current centroids:*

```
while (term_code == 0) {
  # Compute Euclidean squared distances records-centroids
  D = -2 * (X %*% t(C)) + t(rowSums(C ^ 2));
  # Find the closest centroid for each record
  P = (D <= rowMins(D));
  # If records belong to multiple centroids, share them
  P = P / rowSums(P);
  # Compute the column normalization factor for P
  P_denom = colSums(P);
  # Compute new centroids as weighted averages
  C_new = (t(P) %*% X) / t(P_denom); # ...
}
```

The first matrix multiplication $\mathbf{X} \mathbf{C}^\top$ yields another federated, row-partitioned matrix. The subsequent row aggregates and element-wise operations similarly create aligned federated intermediates, which are then aggregated and only consolidated in aggregate form via $\text{colSums}(\mathbf{P})$ and $\mathbf{P}^\top \mathbf{X}$, where the latter is an aligned matrix multiplication of two federated matrices. Note that this built-in function script is agnostic of local, distributed, or federated input matrices.

While some ML algorithms directly map to federated operations that preserve private data, other algorithms need dedicated compiler assistance for generating valid runtime plans. Specifying data exchange (i.e., privacy) constraints for federated raw data, tracking derived properties of intermediates and data transfers, and generating constraint-aware plans is an important direction for future work but beyond the scope of this thesis.

5.2.3 Federated Data Preparation

Besides ML-based data cleaning and data pre-processing—which are based on federated linear algebra—there are other data preparation techniques that require special federated support. These operations include feature transformations, and data access methods for raw data. In contrast to typically stateless ML systems and libraries, the architecture of standing federated workers further provides rich opportunities for reuse and adaptive data reorganization across multiple pipeline runs of a single user and multiple tenants.

Feature Transformations: The `transformencode` and `transformapply` standard feature transformations already presented in [Chapter 4](#) like recoding, feature hashing, binning, and one-hot encoding also apply to the federated setting. The federated instructions of these operations leverage the flexibility of UDFs via `EXEC_UDF` and preserve privacy of the raw federated data. In detail, federated `transformencode` uses a two-pass approach. First, we build encoder-specific metadata for all non-pass-through features (i.e., all columns except unmodified numeric columns) at the federated sites, as well as consolidate—and optionally sort—the metadata for consistent encoding. Second, we broadcast the aggregated metadata, and in a second pass over the federated data, then perform the actual encoding. The outputs are a federated encoded matrix with consistently-aligned one-hot-encoded features (equivalent to local encoding), and a local metadata frame.

Advanced Federated Feature Transformations: There are many opportunities for improved federated feature transformations. First, techniques like zigzag joins [226]—that rely on Bloom filters for pre-filtering—can be adapted for determining categories that need to be exchanged with the coordinator, thereby reducing data transfer and revealed information. Similarly, for features that only exist at a single federated site (e.g., column-partitioned federated data), we only need to exchange the number (instead of the set) of distinct items. Second, there is a tradeoff between privacy versus accuracy. Instead of recoding, users can resort to feature hashing (with an agreed hash function), which is computed in a purely federated manner without data exchange. However, hash collisions merge multiple categories into one feature, which might negatively affect accuracy. In our current implementation, we support the different transformations but leave the choice up to the user because we expect related negotiations among involved parties.

5.2.4 ExDRA Future Work

The ExDRA paper [24] highlighted three different directions of further improvements on the federated backend: lineage-based reuse [190], compression [74], and incremental maintenance [172, 205]. While generally applicable, there is a big potential in leveraging these for the standing federated sites that allow asynchronous optimizations between federated requests. For exploratory ML pipelines with repeated raw data access and data enrichment, techniques such as workload-aware compression would allow to eliminate unnecessary redundancy, and specialize the data representation for the observed workload characteristics, while retaining the appearance of a stateless ML system and preserving the privacy of the federated raw input data.

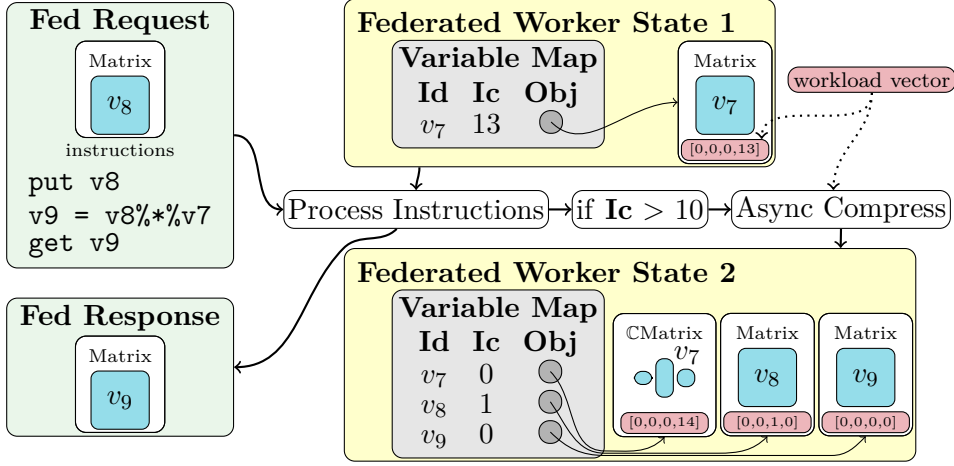


Figure 5.3: Single Request Asynchronous Compression State-change Overview.
Ic is the instruction counter, **Obj** is an Object Pointer

5.3 Standing Federated Workers

Since the ExDRA paper [24], we have updated the federated backend of SystemDS. One addition is adaptive workload compression on standing workers. It consists of two components: workload collection and asynchronous compression. Figure 5.3 shows a high-level example of the updated process.

Dynamic Collection of Workload: Once a request arrives at the federated site, it contains a sequence of instructions to be performed on intermediate variables and data missing to execute the instructions. The federated site contains an execution context with all live variables. If workload adapting compression is enabled, we maintain a ConcurrentHashMap that maps all intermediate values to workload vectors. The workload vectors are equivalent to the workload vectors extracted in AWARE (Figure 3.11). For simplification, Figure 5.3 uses smaller workload vectors with fewer asymptotic buckets. When processing individual instructions, each input increments its corresponding workload vectors and instruction counters. A single federated request can contain multiple instructions. Therefore, the workload vectors and instruction counters may increase many times on one request. In the example, an LMM instruction uses v_7 , increasing its corresponding bucket from 13 to 14. New intermediates or transferred variables also get assigned a workload vector. Therefore, v_8 gets a new workload vector and increases its RMM counter, while the result v_9 contains an empty workload. Depending on the instructions, the result requested in a federated instruction could be a compressed intermediate. In such cases, the result would be transferred in a compressed form.

Asynchronous Compression: Once the entire federated request is processed, we spawn a thread to loop through the variable map and identify intermediates that should be analyzed. If a variable's instruction counter is above ten (the default, but another threshold can be provided by the user), we reset the counter to zero and spawn an additional asynchronous task to try to analyze, compress, and/or morph that intermediate. This design allows the federated site to return the results of requests as fast as possible and leverage the idle time after requests for compression. In the Figure 5.3, we see that the v_7 variable has an instruction count of 13. Once the federated request is processed, the asynchronous analysis identifies v_7 as a candidate and reset the instruction counter. The final state of the federated worker, after the asynchronous analysis, is v_7 in a workload-optimized compressed format. Because the compression is performed asynchronously out of the critical path of the instruction execution, it does not directly impact execution time, except in cases where other federated requests arrive while performing analysis and compression.

5.4 Experiments

This section describe the experimental results from the ExDRA paper [24]. Our experiments study the performance of the described federated runtime backend of SystemDS, with various ML algorithms and pipelines, network configurations, and in comparison with local execution and other ML systems. Since the submission of the paper, all results have been independently fully reproduced¹.

5.4.1 Experimental Setting

Baselines: For evaluating the characteristics of federated linear algebra and parameter servers in controlled yet practically relevant scenarios, we compare the following four main baselines:

- *Local:* Our main baseline is SystemDS with local, in-memory operations, which uses equivalent runtime plans and runtime operations, but executed locally on a single node.
- *Federated LAN:* The federated runtime backend dispatches runtime operations on federated matrices to the described federated linear algebra operations and parameter server. For Federated LAN, all coordinator and workers nodes are part of a local area network (LAN) of two racks, connected via an HPE FlexFabric5710 48XGT switch.
- *Federated WAN:* In addition to Federated LAN, we experiment with the federated backend in a wide-area network (WAN) setting. Here, a client node runs the coordinator in Copenhagen, Denmark and the workers run in a cluster (described below) in Graz, Austria – a distance of more than 1,000 km with round-trip latency of about 35-60 ms, and data transfer bandwidth of about 1.4-2 MB/s.
- *Other ML Systems:* To ensure that *Local* is a competitive baseline, we also compare with local execution in Scikit-learn 0.23 [186] for traditional batch ML algorithms, and TensorFlow 2.3.1 [5] for mini-batch neural network workloads. These systems do not support federated ML.

Cluster Configuration: We ran all experiments described here on eight nodes, each having a single AMD EPYC 7302 CPU at 3.0–3.3 GHz (16 physical/32 virtual cores), 128 GB DDR4 RAM at 2.933 GHz balanced across 8 memory channels, 2 × 480 GB SATA SSDs (system), 12 × 2 TB SATA HDDs (data), and 2 × 10Gb Ethernet. The nominal peak performance of each node is 768 GFLOP/s and 183.2 GB/s, whereas we measured 109.6 GB/s for an 8 GB matrix-vector multiplication. For wide-area network tests, we use an additional client node Dell XPS 15 with one Intel i9-9980HK CPU at 2.4–5.0 GHz (8 physical/16 virtual cores), and 32 GB DDR4 RAM at 2.666 GHz. Our software stack comprises Ubuntu 20.04.1 as operating system, OpenJDK Java 1.8.0_265, and SystemDS 2.0.0++ (as of 03/2021), configured with native Intel MKL BLAS for dense matrix-matrix multiplications. The coordinator and worker nodes use consistent JVM configurations of `-Xmx110g -Xms110g -Xmn11g`, while the WAN client uses `-Xmx30g -Xms30g -Xmn3g`.

Workloads: The tested workloads include the ML algorithms linear regression (LM), L_2 -regularized support vector machine (L2SVM) and multi-class logistic regression (MLogReg) for classification, K-Means for clustering (with K=50 centroids), principal component analysis (PCA) for dimensionality reduction (with K=10 projected features), as well as two parameter server models: a fully-connected feed-forward network (FFN) with BSP, 5 epochs, batch size 512, and trained with stochastic gradient decent (SGD) with Nesterov momentum, as well as a convolutional neural network (CNN) with BSP, 2 epochs, batch size 128, and standard SGD. These algorithms are trained on a synthetic $1\text{M} \times 1,050$ feature matrix (after one-hot encoding categorical features), which closely resembles the characteristics of the data from a paper

¹<https://github.com/damslab/reproducibility/tree/master/sigmod2021-exdra-p523>

production use case [24]. For the CNN scenario though, we use the standard 60K/10K \times 784 MNIST dataset from computer vision. The feature matrix \mathbf{X} is stored as a row-partitioned, federated matrix with balanced partition sizes at the federated sites (i.e., worker nodes), while the labels \mathbf{y} are stored at the coordinator node. We fix the number of maximum iterations for iterative ML algorithms and report the end-to-end runtime—including JVM startup and I/O from binary files—as a mean of (at least) three repetitions.

5.4.2 ML Algorithms Performance

In a first set of experiments, we compare the local ML algorithms performance with both Federated LAN and WAN. We also vary the number of federated workers, evaluate communication settings such as SSL encryption, and compare other ML systems.

ML Algorithms: Figure 5.4 shows the end-to-end runtime of the ML algorithms. As a first step, consider a scenario of three federated workers (the number of workers is varied on the x-axis), which require additional communication but also provide more computational resources. The ML algorithms have different characteristics in that regard. First, LM internally calls an iterative conjugate-gradient LM method (used for $\text{ncol}(\mathbf{X}) > 1,024$), where each iteration performs an $\mathbf{X}^\top(\mathbf{X}\mathbf{v})$ over the federated data. Compared to local, we observe low overhead and already a runtime improvement with three workers. The Fed LowerBound represents the remaining local execution time that is not subject to federated computation and thus, the best Fed LAN could achieve. Second, L2SVM uses two nested while loops, where each outer iteration computes gradients, and the inner loop performs a line search along the gradient. Since the federated \mathbf{X} is only accessed via matrix-vector and vector-matrix operations in the outer loop, the differences to the local runtimes are much smaller. Third, MLogReg also uses two nested while loops, but each inner iterations performs an $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$ on the federated \mathbf{X} and accordingly, we see again a solid improvement with three workers. Fourth, a single run of K-Means has a single while loop, which uses more compute-intensive matrix-matrix multiplications. Fifth, PCA is a non-iterative algorithm and computes an Eigen decomposition of $\mathbf{X}^\top\mathbf{X}$ and subsequently, projects the data via another matrix multiplication to $K=10$ features. With large number of rows, the two matrix multiplications dominate the runtime. Both K-Means and PCA accordingly show also substantial improvements compared to local execution. Finally, FNN and CNN use the mini-batch parameter server architecture with local per-batch updates and global per-epoch synchronization. The larger compute resources of the federated backend yield improvements despite the additional communication. Most importantly, none of these federated ML algorithms ever communicates the raw input data to the coordinator (and thus, preserve privacy of the federated data), they all show only small overhead, and in many cases even runtime improvements. In additional experiments with federated labels \mathbf{y} and/or smaller number of columns (not shown here), we observe that some algorithms like L2SVM incur substantially larger overhead though, because all vector operations of the inner loop are then converted to federated operations as well, which increases communication latency without benefiting from the larger computational resources. In the Federated WAN setting, the relative communication overhead is also substantially higher, but even there, the end-to-end overhead is moderate, which renders federated learning practical for real deployments.

Scalability: Besides the comparison of Local, Federated LAN, and Federated WAN, Figure 5.4 also shows the scalability of our federated backend with increasing number of federated workers. We investigate strong scaling behavior by keeping the data size constant. The coordinator sends federated requests in parallel to all the workers and either broadcasts all side inputs or only relevant slices according to operation requirements. The size of communicated intermediates is moderate in all scenarios though; typically, we exchange only vectors in the number of rows (LM, L2SVM, MLogReg, KMeans) or columns (LM, L2SVM) of \mathbf{X} , columns-by-classes (MLogReg), columns-by-centroids (K-Means), columns-by-columns (PCA), or model sizes (FFN, CNN). Accordingly, we see good scalability, where additional

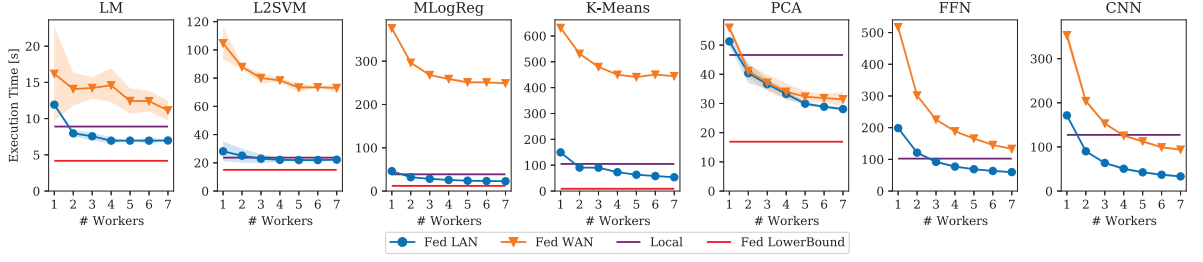


Figure 5.4: Algorithm Comparison and Scalability with Number of Federated Workers ($10^6 \times 1,050$ feature matrix \mathbf{X}).

workers even improve the runtime up until a point, where the partitions per worker become so small that communication increasingly dominates the total runtime. For L2SVM and LM, the improvements are smaller because L2SVM is dominated by vector operations at the coordinator, and LM has a very small runtime, where initial startup constitutes a large fraction of total execution time. In the Federated WAN, the communication overhead is larger but still moderate overall. As the number of workers increases both federated computation and—maybe surprisingly—communication time reduces. The coordinator sends RPCs to all workers in parallel (which mitigates the additional latency), and the more workers the smaller some of the transferred intermediates (e.g., $n/\text{\#workers}$).

SSL Encryption: The federated backend of SystemDS supports SSL-encrypted communication channels between the coordinator and federated workers. We leverage Netty’s `SslContext` for encrypting the federated requests and responses including exchanged data. In a next experiment, we study the overhead this encrypted communication entails. Figure 5.5

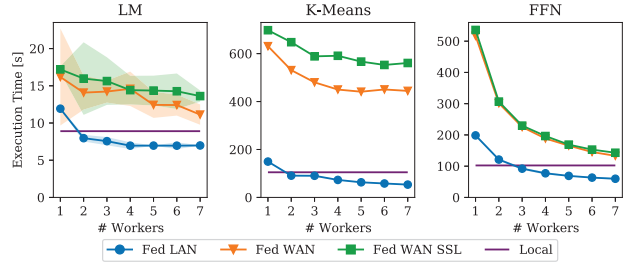


Figure 5.5: Comparison of Communication Settings.

compares LM, K-Means, and FFN—which have very different characteristics and thus, showed different scaling behavior—in the Federated LAN, Federated WAN, and Federated WAN with SSL settings. For LM—where exchanged intermediates are small (vectors in the number of columns)—the overhead of WAN and additional SSL encryption is limited to about 2x and 10%, respectively. K-Means shows larger overhead of about 4-8x in a WAN setting due to more iterations and larger transfers (columns-by-centroids), and again about 15% overhead for SSL. In contrast, the federated parameter server shows only moderate WAN and SSL overhead because of the higher computational workload per worker and infrequent per-epoch global model updates and synchronization.

ML System Comparison: With the comparison of local and federated algorithms in mind, we can now turn to a comparison with other ML systems, specifically Scikit-learn [186] and TensorFlow [5] as widely-used ML systems. We select K-Means, PCA, FFN, and CNN for comparison in order to limit the influence of algorithmic differences. The algorithms were configured to yield a similar number of iterations (e.g., K-Means) and final accuracy. Figure 5.6 shows the results comparing local and Federated LAN configurations of K-Means and PCA with Scikit-learn, and FFN and CNN with TensorFlow.

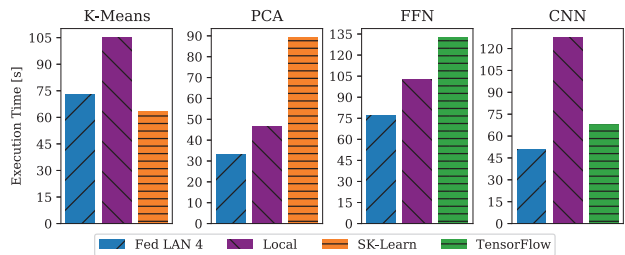


Figure 5.6: Comparison with Other ML Systems.

The algorithms were configured to yield a similar number of iterations (e.g., K-Means) and final accuracy. Figure 5.6 shows the results comparing local and Federated LAN configurations of K-Means and PCA with Scikit-learn, and FFN and CNN with TensorFlow.

Overall, we observe mixed results. K-Means is 1.6x slower than Scikit-learn, while PCA is 2x faster. Similarly, FFN is 25% faster, while CNN is 2x slower than TensorFlow. We attribute these differences to remaining algorithmic discrepancies, and the comparison with best-of-breed ML systems for the different algorithms, whereas SystemDS aims to support a wide range of algorithms and deployments. For CNN, the overhead is partially due to SystemDS using sparse 2d backwards convolution data/filter and other operations because MNIST and related intermediates are just below the internal sparsity threshold. Moreover, TensorFlow’s parallel operator scheduling is advantageous in small mini-batch scenarios. In additional experiments on a spectrum of data characteristics, we observed relative improvements of SystemDS compared to the other ML systems with increasing sparsity, number of rows, and batch size. Most importantly, these comparisons ground the observed Federated LAN results in a performance range close to state-of-the-art systems, supporting the conclusion of applicability in practice.

5.4.3 ML Pipelines Performance

In a second set of experiments, we now return to our main motivation of supporting entire ML pipelines on federated raw input data without central data consolidation.

ML Pipeline Setup: The workload is a simplified training pipeline of a paper production use case [24]. This pipeline reads the input data of continuous and categorical features as a federated frame, and transforms the frame via recoding and one-hot encoding into a numeric input matrix and a meta frame that holds the recode maps. Subsequently, we perform value clipping for values outside the interval $[-1.5\sigma, 1.5\sigma]$ of column standard deviations, normalize the data to zero column means and column standard deviations one, and finally, create 70/30 train and test splits. In order to retain a balanced data distribution across federated workers, we perform this splitting via a uniformly sampled selection-matrix-multiply. Finally, we train a regression or neural network model on the train split, evaluate its performance on the test split, and write out the model and metadata.

Scalability: Figure 5.7 shows the total execution time of ML pipeline P2 on a synthetic federated dataset of 10^6 observations that map—after encoding—to a $1M \times 1,050$ feature matrix. As the number of workers increases, we again see good improvements compared to local operations. The federated transformencode, pre-processing like outlier removal and normalization, train/test splitting, and

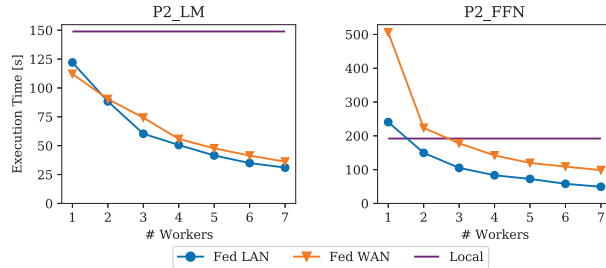


Figure 5.7: ML Pipeline Scalability.

LM training nicely map to federated linear algebra operations. P2_LM and P2_FFNN differ only in the used training algorithm. The larger compute workload of P2_FFNN then explains the better scalability with more workers. For P2_LM, already a single worker shows improvements over local execution because of the additional resources of a coordinator and one worker compared to a single node, which can be used for garbage collection and JIT compilation. Finally, we also partially support the remaining ML pipelines of our use cases in a federated environment. These pipelines include pre-processing steps like missing value imputation, PCA, correlation matrices, density-based clustering, as well as the task-parallel training of multiple GMM (Gaussian Mixture Model) instances.

5.4.4 Asynchronous Compression

To combine the topics of workload-aware compression and federated learning, this section describes the results of running workload-aware federated workers.

Table 5.2: K-Means Scaling Results ($runs = 10$, $k = 10$) on the Adult Dataset.

Scale	ULA - Workers	AWARE - Workers
1x	9.4 sec	9.4 sec
10x	16.6 sec	15.6 sec
100x	140.3 sec	97.1 sec

Experimental Setting: The subset of the nodes used in AWARE’s [23] distributed experiment. In detail, we use a cluster of 1+2 nodes, each with a single AMD EPYC 7443P CPU at 2.85 GHz (24 physical/48 virtual cores) and 256 GB DDR4 RAM at 3.2 GHz. The experiment is set up using two federated sites and one controller contained in the same cluster with high-speed 200Gb interconnects. The system runs Ubuntu 20.04.1, OpenJDK java 17.0.11 with JVM arguments `-Xmx250g -Xms250g -Xmn25g -add-modules=jdk.incubator.vector`, and SystemDS version 3.3*, with custom modifications for leveraging Java’s Vector API for compute-intensive operations such as matrix multiplication.

Data: As dataset, we use the Adult dataset [26]. The dataset is preprocessed into a matrix and replicated onto each of the two federated worker sites.

Experiments: We use two experiments to highlight the power of standing workload-aware compressing workers. The first uses the K-means algorithm, while the second uses a chain of right matrix multiplications with interleaved matrix scalar binary additions.

End-to-end K-means: We use K-means as an example algorithm because most operations are pushed to the federated site (as the red line in Figure 5.4 shows). Therefore, the workload-aware federated workers get varied federated requests with various instruction types that they can use to compress their data partitions. Table 5.2 shows the end-to-end results of running K-means clustering. We can observe that on the short-running job with the standard size of Adult, the performance is equivalent. When we increase the data size on the federated site, the workload-aware workers can exploit the data redundancy and achieve a speedup. Figure 5.8 shows the execution time distribution for individual instructions on the federated workers for the 100x scaling experiment. On the right, in red, the normal federated worker shows that especially matrix multiplication (`ba+*`) and the densifying addition (`+`) use a significant portion (51%) of the federated execution time. On the other side, the two federated workers, in blue, choose different compression schemes for their data. The sites choose different schemes because their data partitions have slightly different characteristics. For instance, the first federated worker chooses a scheme with faster right matrix multiplication vs slower overlapping decompression. However, in the end, the total time is roughly the same for both federated sites. The total time sum for the federated sites is larger than the overall time of the algorithm because we ran ten k-means concurrently. Hence, the experiment highlights that the asynchronous workload-aware compression on the federated workers can improve execution performance, even with overlapping requests.

AWARE Worker 1				AWARE Worker 2				Normal ULA Worker			
#	Instruction	Time(s)	Count	#	Instruction	Time(s)	Count	#	Instruction	Time(s)	Count
1	<=	34.618	142	1	ba+*	44.192	284	1	ba+*	85.218	284
2	ba+*	31.332	284	2	<=	26.573	142	2	+	59.181	142
3	r'	30.808	143	3	r'	21.563	143	3	<=	46.958	142
4	+	18.552	142	4	*	14.543	142	4	/	26.407	141
5	*	15.458	142	5	+	13.245	142	5	*	26.398	142
6	uarmin	9.825	142	6	uarmin	11.348	142	6	r'	18.086	143
7	/	6.522	141	7	/	9.309	141	7	uarmin	5.625	142
8	uack+	2.226	141	8	uark+	4.185	142	8	uack+	5.434	141
9	uark+	2.075	142	9	uack+	3.258	141	9	uark+	4.599	142
10	uasqk+	0.482	1	10	uasqk+	0.504	1	10	uasqk+	0.476	1
11	ucumk+	0.426	1	11	==	0.325	1	11	==	0.337	1
12	==	0.380	1	12	ucumk+	0.311	1	12	ucumk+	0.267	1
13	rmvar	0.080	1280	13	rmvar	0.072	1280	13	rmvar	0.080	1280
TOTAL: 152.784 sec				TOTAL: 149.428 sec				TOTAL: 279.066 sec			

Figure 5.8: Instruction Breakdown of Two Workload-aware Workers, and One ULA Worker.

Pseudo Neural Network: The second experiment workload is close to what three fully connected layers in a neural network would use in inference but without activation functions. The specific workload has good properties for compressed linear algebra and potential impact on neural network workloads. However, nonlinear activation functions, a critical part of neural networks, should be added to improve applicability.

Code: The code used for the experiment is a few lines of declarative linear algebra in the form of DML code. Note that the code does it specify that we are using federated instructions.

```
X = read($1)

R = rand(rows = ncol(X), cols = 512, seed = 132)
R2 = rand(rows = 512, cols = 256, seed = 132)
R3 = rand(rows = 256, cols = 128, seed = 132)

for(i in 1:100){
  t1 = time()
  X_r = (((X %%% R +2) %%% R2 +2) %%% R3) +2
  X_m = sum(X_r)
  t2 = time()
  s = toString(X_m, rows=1, cols= 1)
  print(((t2 - t1) / 1000000) + "," + s) # Print time & sum
}
```

Instead of using the available federated data creation, we can dynamically detect that it is a federated workflow based on the `read($1)` command's data. If the data path on disk contains a JSON object specifying the federated address, ports and paths for the federated data. We automatically compile and use appropriate federated instructions. Such a federated JSON file looks like this:

```
[{"dataType": "MATRIX",
  "address": "localhost:8001",
  "filepath": "data/adult_features.data",
  "begin": [0,0],
  "end": [32561,107]
},{
  "dataType": "MATRIX",
  "address": "localhost:8002",
  "filepath": "data/adult_features.data",
  "begin": [32561,0],
  "end": [65122,107]}]
```

Results: Figure 5.9 shows the performance of executing the script with the two federated locations hidden behind a port-forwarded SSH connection binding the local-host ports to the two remote sites. The y-axis is the log-scaled time in milliseconds of a roundtrip of a single for-loop iteration of the DML code, while the x-axis is the iteration count of the loop. The first loop iteration

sends the three random dense matrices, R , to the federated sites. Then, the federated sites compute X_r via the chain of matrix multiplications, followed by summing the result matrix in X'_m , which is the partial sum of each federated site. The sum, X_m , is then calculated by the coordinator by collecting and adding all X'_m from the federated sites. This first iteration is expensive, taking over 2000 ms, because of transfer overheads of the R matrices and JIT compilations from repeated pipeline execution on the federated workers not having effect yet. The second iteration intelligently skips transferring the R matrices because they are already

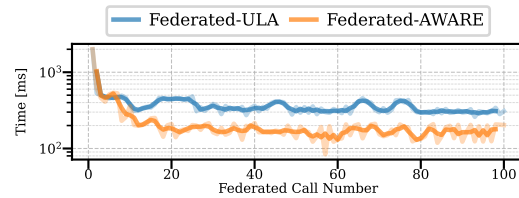


Figure 5.9: Federated AWARE (Adult)

federated. The skip (together with JIT compilation starting to kick in) makes the iterations fall to 400-500 ms for both the workload-aware compressed operations and the baseline ULA execution. We see a slight increase after 3-4 calls in the AWARE execution time to 550 ms, which happens because the asynchronous workload compression has started and incurs a slight overhead to performance. After eight iterations, the ULA and AWARE execution-time diverge because the federated matrix is replaced by the workload-aware compressed version. In the end, after many more calls, the two implementations somewhat stabilised, with ULA at 300ms and AWARE at 190ms. However, these results are with each federated site only performing a small workload with a small dataset.

Scaling: Once we scale the dataset size on each federated site, we see the potential of compressed linear algebra. Figure 5.10 shows the performance of executing the same workload with the Adult dataset 10x and 100x replicated on each of the federated sites. The replication is simply appending the same dataset to itself (`rbind`). While the compression scheme could be designed to exploit this naive replicated data, we do not. The results show the promising and expected behavior, that the compressed operations scale according to the distinct properties of the underlying replicated data, not the increase in data size. The scaling, therefore, does not change the execution time of AWARE once the asynchronous compression has finished.

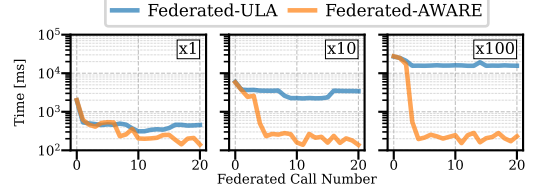


Figure 5.10: Federated AWARE Scaling

5.5 Summary

This chapter introduced the federated backend of SystemDS. We described privacy models, system architecture, and features of the compiler and runtime execution of the federated backend. The results show promising properties of the federated execution model. Extending the standing federated workers with workload analysis and adaptation of asynchronous compression further improves the performance and shows another use case of workload-aware compressed linear algebra. In conclusion, leveraging a declarative language that only defines a program's behavior allows us to automatically compile complex federated learning pipelines that leverage workload-adapting compression behind the covers without changing a single line of code compared to local execution.

6

Conclusions and Future Work

The thesis concludes with a summary, key findings, and suggestions for future research.

6.1 Summary

This thesis confirms that compression is effective at fitting data in available memory, reducing I/O across the storage-memory-cache hierarchy, and increasing instruction parallelism. To get these gains, we introduce a framework integrated into Apache SystemDS [31] for workload-aware compressed linear algebra. Our framework contains new compressed schemes, algorithms, transformations, and direct operations without decompression.

[Chapter 1](#) introduced the topic and highlighted four motivational factors for using compressed linear algebra. First, real-world data contains redundancy, and compression can exploit redundancy. Second, lossy transformations and feature engineering can introduce redundancy in seemingly random data. Third, ML pipelines contain the same redundancy patterns through multiple stages of feature transformations. The chapter concludes that systems should exploit the redundancy and avoid rediscovering compressible patterns through pipeline stages.

[Chapter 2](#) described different lossless compressed formats and how frameworks use compound variations of compression techniques. The variations aim at different tradeoffs between compression size and (de)compression time. The section continued describing related frameworks' support for pushing computation into compressed formats. We identified a common weakness of related work relying on sparse linear algebra.

[Chapter 3](#) showed how workload-aware compression can construct a compressed format of numeric matrices optimized for a specific workload. Unlike previous work, our format allows densifying operations and returns compressed intermediates on more linear algebra operations.

[Chapter 4](#) defined ways of exploiting data redundancy via compressed formats through feature transformations and feature engineering. Furthermore, it contains methods for morphing already compressed representations into other workload-optimized formats without decompression. We show how morphing instructions dynamically compile into directed acyclic graphs of operators to be used at program runtime.

Finally, [Chapter 5](#) contained a description of the federated backend of SystemDS and used it as a use case of applying workload-aware compression to standing federated workers. The workload-aware workers can, via extensions, adapt internal data structures to their workloads.

6.2 Conclusions

There are many conclusions to draw from our work. However, we would like to focus on three:

- First, declarative language abstractions only defining linear algebra operations intentions allow efficient use of specialized data structures. Using the declarative language, we can automatically morph data into workload-optimized formats.
- Second, we conclude that workload-aware compression that summarizes workload and optimizes compression for minimal execution time is better than a pure focus on compression ratios.
- Third, we, via simple lightweight compression techniques, improve the asymptotic behavior of linear algebra operations, and in turn algorithms, by exploiting compressed operations that maximize the reuse of compressed index structures.

6.3 Future Work

New column groups: There is a potential in developing new column groups exploiting the techniques proposed in TOC [147] and GLA [77]. The new groups should be modified techniques that support fused FOR compression and overlapping states similar to AWARE [23]. Both techniques can be modified for these additional functionalities. However, one limitation to overcome is these schemes rely on compressing both the row and column dimensions. Therefore, in the column group setting of AWARE, they would only apply in cases where we co-code many columns. If it works, we should see performance improvements where column groups' performance could scale in the number of grammar rules [77] or LZW entries [147], which is lower than number of distinct tuples many operations in AWARE. Another interesting direction for compressed processing is via functional approximations [169], where operations can be pushed into the function definitions of the compressed format. In a simple example, a polynomial function could approximate the column values. Then, when performing RMM, instead of decompressing, the result would be a modified polynomial function.

Image Augmentations: Another direction of future work is modality-specific augmentation. The contributions in BWARE focused on tabular transformations and non-linear transformations, both crucial components in data-centric ML pipelines. However, missing is image augmentation. Affine transformations performing linear transformations of images have potential in compressed linear algebra. Coincidentally, affine transformations can be performed via an RMM and vector addition, resulting in a compressed input and output from transformations. Furthermore, depending on the interpolation algorithm selected [84], the output of the RMM can be non-overlapping because each column output only contains values from one input column group.

Learned Quantization: Actively making intermediates compressible via, for instance, learning quantization parameters while fitting neural networks. Wiedemann et al. [241] showed that compressing layers on quantized and pruned networks has high compression potential while only leveraging an OLE-like compression scheme. Turning the objective around and instead fitting a model while integrating the compressibility and compressed inference performance in the loss function potentially reduces the cost of neural network operations.

Compressed SIMD Linear Algebra: We developed the compression framework with limited to no exploitation of SIMD. However, vectorized operations directly on the compressed representation have potential. Even with the partially random access patterns from the indirections of the index structures of the compression schemes, SIMD applies via gather and scatter operations. These would apply to DDC, SDC, CONST, RLE, OLE, and FOR groups. Unfortunately, vectorized operations do not apply to GLA and TOC-like techniques without modifications because of the sequential iteration and exploitation of rule sequences.

Hardware Acceleration: Further reducing the execution time we could move the computation of compressed linear algebra to hardware accelerators, such as GPUs. A limitation of moving the implementation to GPU-based execution is the irregularity of the compressed formats, where many differently encoded groups pose a problem. Therefore, a GPU-based implementation of AWARE would have to consider other parameters, such as compression variety, in the workload-aware planning and compression. However, a good reason for hardware-accelerated CLA is that many operations use standard linear algebra instruction types. Additionally, there is no branching behavior in most of the compressed linear algebra operations, and each cell has to be processed similarly. For instance, compressed matrix multiplications contain pre-aggregation followed by normal matrix multiplication, both computations that would fit on GPU devices.

Computational Storage: In the same direction of hardware specialization, BWARE contained some I/O specialization. However, there is more potential in the storage direction leveraging computational storage [144, 106]. For instance, the compression decisions and operation of data can be pushed down to the storage device itself. Computational storage would enable dynamic refinement of the stored format subject to workload characteristics. Rarely used data could, for instance, be heavily compressed, while commonly used data would be readily available. Furthermore, with this adaptive approach, we could co-design the stored format with linear algebra instruction primitives that could be executed while reading the data into system memory.

Streaming Use Cases: The contributions in this thesis focus on full dataset compression to allow exploiting the redundancy of entire columns. TOC [147] targeted the mini-batch use case by compressing individual batches of data. Similar to TOC, there is potential in exploring more compressed linear algebra on streaming data collections [98]. Future work includes redundancy-exploitation of continuous streams of data batches and operations increasing redundancy while processing, such as feature transformations.

While this thesis demonstrates the potential of workload-aware compression, it serves as a foundation for expanding the application of workload-aware compressed linear algebra to a wide range of high-impact future applications.

Bibliography

- [1] Daniel Abadi et al. *The Design and Implementation of Modern Column-Oriented Database Systems*. 2013. ISBN: 1601987544. URL: <https://dl.acm.org/doi/10.5555/2602024>.
- [2] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. “Column oriented Database Systems”. In: *PVLDB* 2.2 (2009), pp. 1664–1665. DOI: [10.14778/1687553.1687625](https://doi.org/10.14778/1687553.1687625). URL: <https://doi.org/10.14778/1687553.1687625>.
- [3] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. “Integrating compression and execution in column-oriented database systems”. In: *SIGMOD*. 2006, pp. 671–682. DOI: [10.1145/1142473.1142548](https://doi.org/10.1145/1142473.1142548). URL: <https://doi.org/10.1145/1142473.1142548>.
- [4] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. “Column-stores vs. row-stores: how different are they really?”. In: *SIGMOD*. 2008, pp. 967–980. ISBN: 9781605581026. DOI: [10.1145/1376616.1376712](https://doi.org/10.1145/1376616.1376712). URL: <https://doi.org/10.1145/1376616.1376712>.
- [5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. “TensorFlow: A System for Large-Scale Machine Learning”. In: *OSDI*. 2016, pp. 265–283. URL: <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi>.
- [6] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. “Impossibility Results for Grammar-Compressed Linear Algebra”. In: *NeurIPS*. 2020. URL: <https://arxiv.org/abs/2010.14181>.
- [7] Behnaz Abdollahi, Naofumi Tomita, and Saeed Hassanpour. “Data Augmentation in Training Deep Learning Models for Medical Image Analysis”. In: *Deep Learners and Deep Learner Descriptors for Medical Applications*. 2020, pp. 167–180. ISBN: 978-3-030-42750-4. DOI: [10.1007/978-3-030-42750-4_6](https://doi.org/10.1007/978-3-030-42750-4_6).
- [8] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. “Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded”. In: *PVLDB* 10.12 (2017), pp. 1849–1852.
- [9] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. “A Survey on Homomorphic Encryption Schemes: Theory and Implementation”. In: *ACM Comput. Surv.* 51.4 (2018), 79:1–79:35. DOI: [10.1145/3214303](https://doi.org/10.1145/3214303).
- [10] Azim Afrozeh, Leonardo X. Kuffo, and Peter Boncz. “ALP: Adaptive Lossless floating-Point Compression”. In: *PACMOD* 1.4 (2023). DOI: [10.1145/3626717](https://doi.org/10.1145/3626717).
- [11] Pankaj K. Agarwal, Sarel Har-Sarel, et al. “Geometric Approximation via Core-sets”. In: *Combinatorial and Computational Geometry*. 2007. URL: <https://api.semanticscholar.org/CorpusID:13812735>.

- [12] Ahmose. *The Rhind Mathematical Papyrus*. https://www.youtube.com/watch?v=g_qbIs1tNmQ. 1550 BC. URL: https://www.britishmuseum.org/collection/object/Y_EA10058.
- [13] Ahmose. *The Rhind Mathematical Papyrus*. © The Trustees of the British Museum, Shared under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) licence. URL: <https://www.britishmuseum.org/collection/image/366139001>.
- [14] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. “Autoscheduling for sparse tensor algebra with an asymptotic cost model”. In: *PLDI*. 2022. ISBN: 9781450392655. DOI: 10.1145/3519939.3523442. URL: <https://doi.org/10.1145/3519939.3523442>.
- [15] Mark A. Aizerman, E. M. Braverman, and L. I. Rozonoer. “Theoretical Foundations of the Potential Function Method in Pattern Recognition Learning”. In: *Automation and Remote Control*. Vol. 25. 1964, pp. 821–837. URL: <https://cs.uwaterloo.ca/~y328yu/classics/kernel.pdf>.
- [16] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. “The Stratosphere platform for big data analytics”. In: *VLDB J.* 23.6 (2014), pp. 939–964. DOI: 10.1007/s00778-014-0357-y. URL: <https://doi.org/10.1007/s00778-014-0357-y>.
- [17] American Statistical Association (ASA). *Airline on-time performance dataset*. <https://stat-computing.org/dataexpo/2009/the-data.html>. 2009.
- [18] G. Antoshenkov, D. Lomet, and J. Murray. “Order preserving string compression”. In: *ICDE*. 1996, pp. 655–663. DOI: 10.1109/ICDE.1996.492216.
- [19] Vincent Arel-Bundock. *Rdatasets: A collection of datasets originally distributed in various R packages*. R package version 1.0.0 <https://raw.githubusercontent.com/vincentarelbundock/Rdatasets/master/csv/carData/Salaries.csv>. 2023. URL: <https://vincentarelbundock.github.io/Rdatasets>.
- [20] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. “Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads”. In: *SIGMOD*. 2016, pp. 583–598. ISBN: 9781450335317. DOI: 10.1145/2882903.2915231. URL: <https://doi.org/10.1145/2882903.2915231>.
- [21] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. “Optimal Column Layout for Hybrid Workloads”. In: *PVLDB*. 2019, pp. 2393–2407. DOI: 10.14778/3358701.3358707. URL: <https://doi.org/10.14778/3358701.3358707>.
- [22] M.A. Bassiouni. “Data Compression in Scientific and Statistical Databases”. In: *IEEE Trans. Softw. Eng.* Vol. SE-11. 10. 1985, pp. 1047–1058. DOI: 10.1109/TSE.1985.231852.
- [23] Sebastian Baunsgaard and Matthias Boehm. “AWARE: Workload-aware, Redundancy-exploiting Linear Algebra”. In: *PACMOD*. Vol. 1. 1. 2023. DOI: 10.1145/3588682. URL: <https://doi.org/10.1145/3588682>.
- [24] Sebastian Baunsgaard, Matthias Boehm, Ankit Chaudhary, Behrouz Derakhshan, Stefan Geißelsöder, Philipp M. Grulich, Michael Hildebrand, Kevin Innerebner, Volker Markl, Claus Neubauer, Sarah Osterburg, Olga Ovcharenko, Sergey Redyuk, Tobias Rieger, Alireza Rezaei Mahdiraji, Sebastian Benjamin Wrede, and Steffen Zeuch. “ExDRa: Exploratory Data Science on Federated Raw Data”. In: *SIGMOD*. 2021, pp. 2450–2463. DOI: 10.1145/3448016.3457549. URL: <https://doi.org/10.1145/3448016.3457549>.

- [25] Sebastian Baunsgaard, Matthias Boehm, Kevin Innerebner, Mito Kehayov, Florian Lackner, Olga Ovcharenko, Arnab Phani, Tobias Rieger, David Weissteiner, and Sebastian Benjamin Wrede. “Federated Data Preparation, Learning, and Debugging in Apache SystemDS”. In: *CIKM*. 2022, pp. 4813–4817. DOI: [10.1145/3511808.3557162](https://doi.org/10.1145/3511808.3557162).
- [26] Barry Becker and Ronny Kohavi. *Adult*. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5XW20>. 1996.
- [27] Souvik Bhattacharjee, Amol Deshpande, and Alan Sussman. “PStore: an efficient storage framework for managing scientific data”. In: *SSDBM*. 2014, 25:1–25:12. DOI: [10.1145/2618243.2618268](https://doi.org/10.1145/2618243.2618268). URL: <https://doi.org/10.1145/2618243.2618268>.
- [28] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. “Dictionary-based order-preserving string compression for main memory column stores”. In: *SIGMOD*. 2009, pp. 283–296. ISBN: 9781605585512. DOI: [10.1145/1559845.1559877](https://doi.org/10.1145/1559845.1559877). URL: <https://doi.org/10.1145/1559845.1559877>.
- [29] L Susan Blackford, Antoine Petit, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. “An updated set of basic linear algebra subprograms (BLAS)”. In: *TOMS* 28.2 (2002), pp. 135–151. DOI: [10.1145/567806.567807](https://doi.org/10.1145/567806.567807). URL: <https://doi.org/10.1145/567806.567807>.
- [30] Davis W. Blalock, Samuel Madden, and John V. Guttag. “Sprintz: Time Series Compression for the Internet of Things”. In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2.3 (2018), 93:1–93:23. DOI: [10.1145/3264903](https://doi.org/10.1145/3264903). URL: <https://doi.org/10.1145/3264903>.
- [31] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. “SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle”. In: *CIDR*. 2020. URL: <https://doi.org/10.48550/arXiv.1909.02976>.
- [32] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. “SystemML’s Optimizer: Plan Generation for Large-Scale Machine Learning Programs”. In: *IEEE Data Eng. Bull.* 37.3 (2014), pp. 52–62.
- [33] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. “SystemML: Declarative Machine Learning on Spark”. In: *PVLDB* 9.13 (2016), pp. 1425–1436. DOI: [10.14778/3007263.3007279](https://doi.org/10.14778/3007263.3007279). URL: <https://doi.org/10.14778/3007263.3007279>.
- [34] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. “On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML”. In: *PVLDB* 11.12 (2018), pp. 1755–1768. DOI: [10.14778/3229863.3229865](https://doi.org/10.14778/3229863.3229865). URL: <https://doi.org/10.14778/3229863.3229865>.
- [35] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas R. Burdick, and Shivakumar Vaithyanathan. “Hybrid parallelization strategies for large-scale machine learning in SystemML”. In: *PVLDB* 7.7 (2014). ISSN: 2150-8097. DOI: [10.14778/2732286.2732292](https://doi.org/10.14778/2732286.2732292). URL: <https://doi.org/10.14778/2732286.2732292>.
- [36] Martin Boissier. “Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems”. In: *PVLDB* 15.4 (2022), pp. 780–793. DOI: [10.14778/3503585.3503588](https://doi.org/10.14778/3503585.3503588). URL: <https://doi.org/10.14778/3503585.3503588>.

- [37] Martin Boissier. “Robust and budget-constrained encoding configurations for in-memory database systems”. In: *PVLDB*. Vol. 15. 4. 2021, pp. 780–793. DOI: [10.14778/3503585.3503588](https://doi.org/10.14778/3503585.3503588). URL: <https://doi.org/10.14778/3503585.3503588>.
- [38] Martin Boissier and Max Jendruk. “Workload-Driven and Robust Selection of Compression Schemes for Column Stores”. In: *EDBT*. 2019, pp. 674–677. DOI: [10.5441/002/edbt.2019.84](https://doi.org/10.5441/002/edbt.2019.84). URL: <https://doi.org/10.5441/002/edbt.2019.84>.
- [39] Martin Boissier and Max Jendruk. “Workload-Driven and Robust Selection of Compression Schemes for Column Stores”. In: *EDBT*. 2019. URL: <https://api.semanticscholar.org/CorpusID:81989532>.
- [40] Keith Bonawitz et al. “Towards Federated Learning at Scale: System Design”. In: *MLSys*. 2019. URL: <https://proceedings.mlsys.org/book/271.pdf>.
- [41] Peter Boncz, Thomas Neumann, and Viktor Leis. “FSST: fast random access string compression”. In: *PVLDB*. 2020. DOI: [10.14778/3407790.3407851](https://doi.org/10.14778/3407790.3407851). URL: <https://doi.org/10.14778/3407790.3407851>.
- [42] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A training algorithm for optimal margin classifiers”. In: *COLT*. 1992, pp. 144–152. ISBN: 089791497X. DOI: [10.1145/130385.130401](https://doi.org/10.1145/130385.130401). URL: <https://doi.org/10.1145/130385.130401>.
- [43] Léon Bottou and Gaëlle Loosli. *The infinite MNIST dataset*. <https://leon.bottou.org/projects/infmnist>. 2007.
- [44] Google Brain. *bfloat16*. 2024. URL: <https://cloud.google.com/tpu/docs/bfloat16>.
- [45] Lars Buitinck, Gilles Louppe, Mathieu Blondel, et al. “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD*. 2013, pp. 108–122.
- [46] Aydin Buluc and John R. Gilbert. “On the representation and multiplication of hypersparse matrices”. In: *IPDPS*. 2008. DOI: [10.1109/IPDPS.2008.4536313](https://doi.org/10.1109/IPDPS.2008.4536313).
- [47] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Eng. Bull.* 38 (Jan. 2015).
- [48] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. “LEA: A Learned Encoding Advisor for Column Stores”. In: *aiDM@SIGMOD*. 2021, pp. 32–35. DOI: [10.1145/3464509.3464885](https://doi.org/10.1145/3464509.3464885). URL: <https://doi.org/10.1145/3464509.3464885>.
- [49] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. “The smallest grammar problem”. In: *IEEE Trans. Inf. Theor.* 51.7 (July 2005), pp. 2554–2576. ISSN: 0018-9448. DOI: [10.1109/TIT.2005.850116](https://doi.org/10.1109/TIT.2005.850116). URL: <https://doi.org/10.1109/TIT.2005.850116>.
- [50] Lingjiao Chen, Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. “Towards Linear Algebra over Normalized Data”. In: *PVLDB* 10.11 (2017), pp. 1214–1225. DOI: [10.14778/3137628.3137633](https://doi.org/10.14778/3137628.3137633). URL: <https://doi.org/10.14778/3137628.3137633>.
- [51] Yann Collet and Murray Kucherawy. *Zstandard Compression and the 'application/zstd' Media Type*. RFC 8878. Feb. 2021. DOI: [10.17487/RFC8878](https://doi.org/10.17487/RFC8878). URL: <https://www.rfc-editor.org/info/rfc8878>.
- [52] Apache Parquet com. *Apache Parquet: column-oriented data file format designed for efficient data storage and retrieval*. 2023. URL: <https://parquet.apache.org/>.
- [53] Apache Arrow community. *Apache Arrow: A cross-language development platform for in-memory analytics*. 2023. URL: <https://arrow.apache.org/>.
- [54] Apache Arrow community. *Apache Arrow: Dictionary Encoding*. 2023. URL: <https://arrow.apache.org/docs/java/vector.html#dictionary-encoding>.

- [55] Patrick Damme. “Analytical Query Processing Based on Continuous Compression of Intermediates”. In: *PhD Thesis, Technische Universität Dresden*. 2020. URL: <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-723288>.
- [56] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. “Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)”. In: *EDBT*. 2017, pp. 72–83. DOI: [10.5441/002/edbt.2017.08](https://doi.org/10.5441/002/edbt.2017.08). URL: <https://doi.org/10.5441/002/edbt.2017.08>.
- [57] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. “Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses)”. In: *EDBT*. 2017. URL: <https://doi.org/10.5441/002%2Fedbt.2017.08>.
- [58] Patrick Damme, Dirk Habich, and Wolfgang Lehner. “Direct Transformation Techniques for Compressed Data: General Approach and Application Scenarios”. In: *Advances in Databases and Information Systems*. 2015, pp. 151–165. ISBN: 978-3-319-23135-8.
- [59] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. “From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms”. In: *ACM Trans. Database Syst.* 44.3 (2019), 9:1–9:46. DOI: [10.1145/3323991](https://doi.org/10.1145/3323991). URL: <https://doi.org/10.1145/3323991>.
- [60] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. “From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms”. In: *TODS*. Vol. 44. 3. 2019. DOI: [10.1145/3323991](https://doi.org/10.1145/3323991). URL: <https://doi.org/10.1145/3323991>.
- [61] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. “MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model”. In: *PVLDB*. 2020, pp. 2396–2410. URL: <https://doi.org/10.14778/3407790.3407833>.
- [62] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. “MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model”. In: *PVLDB*. Vol. 13. 12. 2020, pp. 2396–2410. DOI: [10.14778/3407790.3407833](https://doi.org/10.14778/3407790.3407833). URL: <https://doi.org/10.14778/3407790.3407833>.
- [63] J. L. Dawson. “Suffic Removal for Word Conflation”. In: *ALLC Bulletin*. Vol. 2. 3. 1974, pp. 33–46. URL: <https://sigir.org/files/museum/pub-21/98.pdf>.
- [64] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc’Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. “Large Scale Distributed Deep Networks”. In: *NeurIPS*. 2012, pp. 1232–1240. URL: <https://proceedings.neurips.cc/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html>.
- [65] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (2008), pp. 107–113. DOI: [10.1145/1327452.1327492](https://doi.org/10.1145/1327452.1327492).
- [66] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibio Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. “The Data Civilizer System”. In: *CIDR*. 2017. URL: <http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf>.
- [67] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. “Optimizing Machine Learning Workloads in Collaborative Environments”. In: *SIGMOD*. 2020, pp. 1701–1716. DOI: [10.1145/3318464.3389715](https://doi.org/10.1145/3318464.3389715). URL: <https://doi.org/10.1145/3318464.3389715>.

- [68] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. 1996. DOI: [10.17487/RFC1951](https://doi.org/10.17487/RFC1951). URL: <https://www.rfc-editor.org/info/rfc1951>.
- [69] Peter Gustav Lejeune Dirichlet. *Schubfachprinzip*. 1834.
- [70] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. “Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management”. In: Mar. 2019. DOI: [10.5441/002/edbt.2019.28](https://doi.org/10.5441/002/edbt.2019.28).
- [71] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: <https://archive.ics.uci.edu/ml>.
- [72] Jarek Duda. “Asymmetric numeral systems as close to capacity low state entropy coders”. In: *CoRR* abs/1311.2540 (2013). arXiv: [1311.2540](https://arxiv.org/abs/1311.2540). URL: <http://arxiv.org/abs/1311.2540>.
- [73] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. “Compressed linear algebra for declarative large-scale machine learning”. In: *Commun. ACM* 62.5 (2019), pp. 83–91. ISSN: 0001-0782. DOI: [10.1145/3318221](https://doi.org/10.1145/3318221). URL: <https://doi.org/10.1145/3318221>.
- [74] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. “Compressed Linear Algebra for Large-Scale Machine Learning”. In: *PVLDB* 9.12 (2016), pp. 960–971. DOI: [10.14778/2994509.2994515](https://doi.org/10.14778/2994509.2994515). URL: <https://doi.org/10.14778/2994509.2994515>.
- [75] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. “Compressed linear algebra for large-scale machine learning”. In: *Vldb J.* 27.5 (2018), pp. 719–744. DOI: [10.1007/s00778-017-0478-1](https://doi.org/10.1007/s00778-017-0478-1). URL: <https://doi.org/10.1007/s00778-017-0478-1>.
- [76] Facebook. *Zstandard*. <https://facebook.github.io/zstd/>. Sept. 8, 2023.
- [77] Paolo Ferragina, Giovanni Manzini, Travis Gagie, Dominik Koppl, Gonzalo Navarro, Manuel Striani, and Francesco Tosoni. “Improving matrix-vector multiplication via lossless grammar-compressed matrices”. In: *PVLDB*. Vol. 15. 10. 2022, pp. 2175–2187. DOI: [10.14778/3547305.3547321](https://doi.org/10.14778/3547305.3547321). URL: <https://doi.org/10.14778/3547305.3547321>.
- [78] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. “Auto-Sklearn 2.0: The Next Generation”. In: *CoRR* abs/2007.04074 (2020).
- [79] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. “Auto-sklearn: Efficient and Robust Automated Machine Learning”. In: *AutoML*. 2019, pp. 113–134.
- [80] Unified Acceleration Foundation. *oneAPI: unified multiarchitecture, multi-vendor programming model with common API’s accross accelerator architectures*. <https://oneapi.io/>. 2024.
- [81] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel Roy, and Michael Carbin. “Pruning Neural Networks at Initialization: Why Are We Missing the Mark?” In: *ICLR*. 2021. URL: <https://openreview.net/forum?id=Ig-VyQc-MLK>.
- [82] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. “FITing-Tree: A Data-aware Index Structure”. In: *SIGMOD*. 2019, pp. 1189–1206. ISBN: 9781450356435. DOI: [10.1145/3299869.3319860](https://doi.org/10.1145/3299869.3319860). URL: <https://doi.org/10.1145/3299869.3319860>.
- [83] Craig Gentry. “Fully homomorphic encryption using ideal lattices”. In: *STOC*. 2009, pp. 169–178.

- [84] *Geometric Image Transformations OpenCV*. 2024. URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html.
- [85] Bogdan Vladimir Ghita, Diego G. Tomé, and Peter A. Boncz. “White-box Compression: Learning and Exploiting Compact Table Representations”. In: *CIDR*. 2020. URL: <https://api.semanticscholar.org/CorpusID:210706292>.
- [86] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. “SystemML: Declarative machine learning on MapReduce”. In: *ICDE*. 2011. ISBN: 9781424489596. DOI: 10.1109/ICDE.2011.5767930. URL: <https://doi.org/10.1109/ICDE.2011.5767930>.
- [87] Ran Gilad-Bachrach, Nathan Dowlan, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: *ICML*. Vol. 48. 2016, pp. 201–210.
- [88] J. Goldstein, R. Ramakrishnan, and U. Shaft. “Compressing relations and indexes”. In: *ICDE*. 1998, pp. 370–379. DOI: 10.1109/ICDE.1998.655800.
- [89] GOOGLE. *Snappy Format description*. https://github.com/google/snappy/blob/main/format_description.txt. 2024.
- [90] GOOGLE. *Snappy: A compression/decompression library Version 1.1.10.3*. <https://google.github.io/snappy/>. 2024.
- [91] Google. *Quantization Aware Training with TensorFlow Model Optimization Toolkit - Performance with Accuracy*. <https://blog.tensorflow.org/2020/04/quantization-aware-training-with-tensorflow-model-optimization-toolkit.html>. 2020.
- [92] Google. *Sparse Tensor*. https://www.tensorflow.org/guide/sparse_tensor.
- [93] Google. *TensorFlow Model Optimization Toolkit - Pruning API*. <https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html>. 2019.
- [94] Kazushige Goto and Robert van de Geijn. “Anatomy of high-performance matrix multiplication”. In: *TOMS* 34.3 (2008), pp. 1–25.
- [95] Stefan Grafberger, Paul Groth, and Sebastian Schelter. “Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines”. In: *PACMOD* 1.2 (2023), 128:1–128:26. DOI: 10.1145/3589273.
- [96] H.L. Gray and W. R. Schucany. *The Generalized Jackknife Statistic*. 1972.
- [97] The HDF Group. *The HDF5 Library and File Format*. <https://www.hdfgroup.org/solutions/hdf5/>. 2021.
- [98] Philipp Marian Grulich. *Query Compilation for Modern Data Processing Environments*. PhD Thesis, Technische Universitaet Berlin (Germany), 2023.
- [99] P. Haas, M. Kandil, A. Lerner, V. Markl, I. Popivanov, V. Raman, and D. Zilio. “Automated statistics collection in action”. In: *SIGMOD*. 2005, pp. 933–935. ISBN: 1595930604. DOI: 10.1145/1066157.1066293. URL: <https://doi.org/10.1145/1066157.1066293>.
- [100] Peter J. Haas and Lynne Stokes. “Estimating the Number of Classes in a Finite Population”. In: *Journal of the American Statistical Association* 93.444 (1998), pp. 1475–1487. DOI: 10.1080/01621459.1998.10473807. eprint: <https://www.tandfonline.com/doi/pdf/10.1080/01621459.1998.10473807>. URL: <https://www.tandfonline.com/doi/abs/10.1080/01621459.1998.10473807>.

- [101] Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. “MorphStore - In-Memory Query Processing Based on Morphing Compressed Intermediates LIVE”. In: *SIGMOD*. 2019, pp. 1917–1920. ISBN: 9781450356435. DOI: [10.1145/3299869.3320234](https://doi.org/10.1145/3299869.3320234). URL: <https://doi.org/10.1145/3299869.3320234>.
- [102] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. “Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores”. In: *PVLDB*. 2012. DOI: [10.14778/2168651.2168652](https://doi.org/10.14778/2168651.2168652). URL: http://vldb.org/pvldb/vol15/p502%5C_felixhalim%5C_vldb2012.pdf.
- [103] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *ICLR*. 2016. URL: <http://arxiv.org/abs/1510.00149>.
- [104] Hazar Harmouch and Felix Naumann. “Cardinality estimation: an experimental survey”. In: *PVLDB* 11.4 (2017), pp. 499–512. ISSN: 2150-8097. DOI: [10.1145/3186728.3164145](https://doi.org/10.1145/3186728.3164145). URL: <https://doi.org/10.1145/3186728.3164145>.
- [105] Ruining He and Julian McAuley. “Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering”. In: *WWW*. 2016, pp. 507–517. DOI: [10.1145/2872427.2883037](https://doi.org/10.1145/2872427.2883037). URL: <https://doi.org/10.1145/2872427.2883037>.
- [106] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. “Delilah: eBPF-offload on computational storage”. In: *DaMoN@SIGMOD*. 2023, pp. 70–76.
- [107] Geoffrey E Hinton and Sam Roweis. “Stochastic Neighbor Embedding”. In: *NeurIPS*. 2002. URL: https://proceedings.neurips.cc/paper_files/paper/2002/file/6150ccc6069bea6b5716254057a194ef-Paper.pdf.
- [108] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. “How to barter bits for chronons: compression and bandwidth trade offs for database scans”. In: *SIGMOD*. 2007, pp. 389–400. DOI: [10.1145/1247480.1247525](https://doi.org/10.1145/1247480.1247525). URL: <https://doi.org/10.1145/1247480.1247525>.
- [109] Mark Horowitz. “Computing’s energy problem (and what we can do about it)”. In: *ISSCC*. 2014. DOI: [10.1109/ISSCC.2014.6757323](https://doi.org/10.1109/ISSCC.2014.6757323).
- [110] Itay Hubara et al. “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations”. In: *JMLR*. Vol. 18. 187. 2018, pp. 1–30. URL: <http://jmlr.org/papers/v18/16-456.html>.
- [111] David A. Huffman. “A Method for the Construction of Minimum-Redundancy Codes”. In: *Proceedings of the IRE*. Vol. 40. 9. 1952, pp. 1098–1101. DOI: [10.1109/JRPROC.1952.273898](https://doi.org/10.1109/JRPROC.1952.273898).
- [112] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. “Database Cracking”. In: *CIDR*. 2007, pp. 68–78. URL: <http://cidrdb.org/cidr2007/papers/cidr07p07.pdf>.
- [113] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. “Updating a cracked database”. In: *SIGMOD*. 2007, pp. 413–424. ISBN: 9781595936868. DOI: [10.1145/1247480.1247527](https://doi.org/10.1145/1247480.1247527). URL: <https://doi.org/10.1145/1247480.1247527>.
- [114] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. “Merging What’s Cracked, Cracking What’s Merged: Adaptive Indexing in Main-Memory Column-Stores”. In: *PVLDB*. 9. 2011. DOI: [10.14778/2002938.2002944](https://doi.org/10.14778/2002938.2002944). URL: <http://www.vldb.org/pvldb/vol14/p586-idreos.pdf>.
- [115] IEEE. *Standard for Floating-Point Arithmetic*. 2008. URL: <https://doi.org/10.1109/IEEESTD.2008.4610935>.

- [116] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. “DeepSqueeze: Deep Semantic Compression for Tabular Data”. In: *SIGMOD*. 2020, pp. 1733–1746. DOI: 10.1145/3318464.3389734. URL: <https://doi.org/10.1145/3318464.3389734>.
- [117] Intel. *Math Kernel Library*. <https://software.intel.com/en-us/intel-mkl/>.
- [118] Olivier Chapelle Jean-Baptiste Tien joycenv. *Display Advertising Challenge*. Data: <https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/>. 2014. URL: <https://kaggle.com/competitions/criteo-display-ad-challenge>.
- [119] Zhanglong Ji, Zachary Chase Lipton, and Charles Elkan. “Differential Privacy and Machine Learning: a Survey and Review”. In: *CoRR* abs/1412.7584 (2014). URL: <http://arxiv.org/abs/1412.7584>.
- [120] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. “SketchML: Accelerating Distributed Machine Learning with Data Sketches”. In: *SIGMOD*. 2018, pp. 1269–1284. DOI: 10.1145/3183713.3196894. URL: <https://doi.org/10.1145/3183713.3196894>.
- [121] Sian Jin, Chengming Zhang, Xintong Jiang, Yunhe Feng, Hui Guan, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. “COMET: A Novel Memory-Efficient Deep Learning Training Framework by Using Error-Bounded Lossy Compression”. In: *PVLDB*. 2022, pp. 886–899. DOI: 10.14778/3503585.3503597. URL: <https://doi.org/10.14778/3503585.3503597>.
- [122] Peter Kairouz, Brendan McMahan, and Virginia Smith. “Federated Learning Tutorial”. In: *NeurIPS*. 2020. URL: <https://slideslive.com/38935813/federated-learning-tutorial>.
- [123] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: 1905.12322 [cs.LG].
- [124] Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. “An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication”. In: *IEEE TPDS*. Vol. 24. 10. 2013, pp. 1930–1940. DOI: 10.1109/TPDS.2012.290.
- [125] Mahmoud Abo Khamis, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, et al. “Learning Models over Relational Data Using Sparse Tensors and Functional Dependencies”. In: *TODS*. Vol. 45. 2. 2020. DOI: 10.1145/3375661. URL: <https://doi.org/10.1145/3375661>.
- [126] Hideaki Kimura, Vivek R. Narasayya, and Manoj Syamala. “Compression Aware Physical Database Design”. In: *PVLDB* 4.10 (2011), pp. 657–668. DOI: 10.14778/2021017.2021023. URL: <https://doi.org/10.14778/2021017.2021023>.
- [127] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. “The tensor algebra compiler”. In: *Proc. ACM Program. Lang.* (2017). DOI: 10.1145/3133901. URL: <https://doi.org/10.1145/3133901>.
- [128] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. “Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks”. In: *NeurIPS*. 2017, pp. 1742–1752. URL: <https://proceedings.neurips.cc/paper/2017/hash/a0160709701140704575d499c997b6ca-Abstract.html>.
- [129] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. “Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA”. In: *J. Mach. Learn. Res.* 18 (2017), 25:1–25:5.

- [130] Kornilios Kourtis, Georgios I. Goumas, and Nectarios Koziris. “Optimizing sparse matrix-vector multiplication using index and value compression”. In: *CF*. 2008, pp. 87–96. DOI: [10.1145/1366230.1366244](https://doi.org/10.1145/1366230.1366244). URL: <https://doi.org/10.1145/1366230.1366244>.
- [131] Alex Krizhevsky et al. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *NeurIPS*. 2012. URL: <https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html>.
- [132] Michael Kuchnik, George Amvrosiadis, and Virginia Smith. “Progressive Compressed Records: Taking a Byte out of DeepLearning Data”. In: *PVLDB*. 2021, pp. 2627–2641. DOI: [10.14778/3476249.3476308](https://doi.org/10.14778/3476249.3476308). URL: <https://doi.org/10.14778/3476249.3476308>.
- [133] Arun Kumar, Jeffrey F. Naughton, and Jignesh M. Patel. “Learning Generalized Linear Models Over Normalized Data”. In: *SIGMOD*. 2015, pp. 1969–1984. DOI: [10.1145/2723372.2723713](https://doi.org/10.1145/2723372.2723713). URL: <https://doi.org/10.1145/2723372.2723713>.
- [134] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. “BtrBlocks: Efficient Columnar Compression for Data Lakes”. In: *PACMOD*. Vol. 1. 2. 2023. DOI: [10.1145/3589263](https://doi.org/10.1145/3589263). URL: <https://doi.org/10.1145/3589263>.
- [135] Harald Lang, Alexander Beischl, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. “Tree-Encoded Bitmaps”. In: *SIGMOD*. 2020, pp. 937–967. ISBN: 9781450367356. DOI: [10.1145/3318464.3380588](https://doi.org/10.1145/3318464.3380588). URL: <https://doi.org/10.1145/3318464.3380588>.
- [136] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *SIGMOD*. 2016, pp. 311–326. DOI: [10.1145/2882903.2882925](https://doi.org/10.1145/2882903.2882925). URL: <https://doi.org/10.1145/2882903.2882925>.
- [137] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *ICML*. 2016, pp. 311–326. ISBN: 9781450335317. DOI: [10.1145/2882903.2882925](https://doi.org/10.1145/2882903.2882925). URL: <https://doi.org/10.1145/2882903.2882925>.
- [138] N.J. Larsson and A. Moffat. “Off-line dictionary-based compression”. In: *Proceedings of the IEEE* 88.11 (2000), pp. 1722–1732. DOI: [10.1109/5.892708](https://doi.org/10.1109/5.892708).
- [139] Robert Lasch, Robert Schulze, Thomas Legler, and Kai-Uwe Sattler. “Workload-Driven Placement of Column-Store Data Structures on DRAM and NVM”. In: *DaMoN@SIGMOD*. 2021, 5:1–5:8. DOI: [10.1145/3465998.3466008](https://doi.org/10.1145/3465998.3466008). URL: <https://doi.org/10.1145/3465998.3466008>.
- [140] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323.
- [141] Daniel Lemire. “Number parsing at a gigabyte per second”. In: *Software: Practice and Experience*. Vol. 51. 8. 2021, pp. 1700–1727. DOI: <https://doi.org/10.1002/spe.2984>.
- [142] Daniel Lemire and Boytsov Boytsov. “Decoding billions of integers per second through vectorization”. In: 2015. DOI: <https://doi.org/10.1002/spe.2203>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2203>.
- [143] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. “Consistently faster and smaller compressed bitmaps with Roaring”. In: *CoRR* abs/1603.06549 (2016). URL: <http://arxiv.org/abs/1603.06549>.
- [144] Alberto Lerner and Philippe Bonnet. *Principles of Database and Solid-State Drive Co-Design*. Springer, 2025.
- [145] Jean Leurechon. *PidgeonHole Principle*. 1624.

- [146] Fengan Li, Lingjiao Chen, Arun Kumar, Jeffrey Naughton, Jignesh Patel, and Xi Wu. “When Lempel-Ziv-Welch Meets Machine Learning: A Case Study of Accelerating Machine Learning using Coding”. In: (Feb. 2017). DOI: [10.48550/arXiv.1702.06943](https://doi.org/10.48550/arXiv.1702.06943).
- [147] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F. Naughton, and Jignesh M. Patel. “Tuple-oriented Compression for Large-scale Mini-batch Stochastic Gradient Descent”. In: *SIGMOD*. 2019, pp. 1517–1534. DOI: [10.1145/3299869.3300070](https://doi.org/10.1145/3299869.3300070). URL: <https://doi.org/10.1145/3299869.3300070>.
- [148] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. “Scaling Distributed Machine Learning with the Parameter Server”. In: *OSDI*. 2014, pp. 583–598. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li%5C_mu.
- [149] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. “PyTorch Distributed: Experiences on Accelerating Data Parallel Training”. In: *PVLDB* 13.12 (2020), pp. 3005–3018. DOI: [10.14778/3415478.3415530](https://doi.org/10.14778/3415478.3415530).
- [150] Panagiotis Liakos, Katia Papakonstantinou, and Yannis Kotidis. “Chimp: Efficient Lossless Floating Point Compression for Time Series Databases”. In: *PVLDB*. 2022, pp. 3058–3070. DOI: [10.14778/3551793.3551852](https://doi.org/10.14778/3551793.3551852). URL: <https://doi.org/10.14778/3551793.3551852>.
- [151] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. “Plato: Approximate Analytics over Compressed Time Series with Tight Deterministic Error Guarantees”. In: *PVLDB*. 2020, pp. 1105–1118. DOI: [10.14778/3384345.3384357](https://doi.org/10.14778/3384345.3384357). URL: <https://doi.org/10.14778/3384345.3384357>.
- [152] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J Elmore. “Decomposed Bounded Floats for Fast Compression and Queries”. In: *PVLDB*. 2021, pp. 2586–2598. DOI: [10.14778/3476249.3476305](https://doi.org/10.14778/3476249.3476305). URL: <https://doi.org/10.14778/3476249.3476305>.
- [153] Chunwei Liu, McKade Umbenhowe, Hao Jiang, Pranav Subramaniam, Jihong Ma, and Aaron J. Elmore. “Mostly Order Preserving Dictionaries”. In: *ICDE*. 2019, pp. 1214–1225. DOI: [10.1109/ICDE.2019.00111](https://doi.org/10.1109/ICDE.2019.00111).
- [154] P. Lockhart. *Arithmetic*. Harvard University Press, 2017. ISBN: 9780674972230. URL: <https://books.google.ch/books?id=W9CbAQAAAJ>.
- [155] Markus Lohrey. “Algorithmics on SLP-compressed strings: A survey”. In: *Groups Complex. Cryptol.* 2012. URL: <https://api.semanticscholar.org/CorpusID:11410540>.
- [156] Shangyo Luo, Dimitrije Jankov, Binhang Yuan, and Chris Jermaine. “Automatic Optimization of Matrix Implementations for Distributed Machine Learning and Linear Algebra”. In: *SIGMOD*. 2021. URL: <https://dl.acm.org/doi/pdf/10.1145/3448016.3457317>.
- [157] Giosuè Cataldo Marinò, Flavio Furia, Dario Malchiodi, and Marco Frasca. “Efficient and Compact Representations of Deep Neural Networks via Entropy Coding”. In: *IEEE Access* 11 (2023), pp. 106103–106125. DOI: [10.1109/ACCESS.2023.3317293](https://doi.org/10.1109/ACCESS.2023.3317293).
- [158] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin

- Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: <https://www.tensorflow.org/>.
- [159] Shirou Maruyama, Masayuki Takeda, Masaya Nakahara, and Hiroshi Sakamoto. “An Online Algorithm for Lightweight Grammar-Based Compression”. In: *2011 First International Conference on Data Compression, Communications and Processing*. 2011, pp. 19–28. DOI: [10.1109/CCP.2011.40](https://doi.org/10.1109/CCP.2011.40).
- [160] Abdelhafid Mazouz, David C. Wong, David Kuck, and William Jalby. “An Incremental Methodology for Energy Measurement and Modeling”. In: *ICPE*. 2017, pp. 15–26. ISBN: 9781450344043. DOI: [10.1145/3030207.3030224](https://doi.org/10.1145/3030207.3030224). URL: <https://doi.org/10.1145/3030207.3030224>.
- [161] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhresch, et al. “Advances in Pre-Training Distributed Word Representations”. In: *LREC*. 2018.
- [162] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *VLDB*. 1998, pp. 476–487.
- [163] Alistair Moffat. “Huffman Coding”. In: *ACM Comput. Surv.* 52.4 (Aug. 2019). ISSN: 0360-0300. DOI: [10.1145/3342555](https://doi.org/10.1145/3342555). URL: <https://doi.org/10.1145/3342555>.
- [164] Payman Mohassel and Yupeng Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: *IEEE Symp. on Security and Privacy*. 2017, pp. 19–38. DOI: [10.1109/SP.2017.12](https://doi.org/10.1109/SP.2017.12).
- [165] Dan Moldovan, James M Decker, Fei Wang, Andrew A Johnson, Brian K Lee, Zachary Nado, D Sculley, Tiark Rompf, and Alexander B Wiltschko. “AutoGraph: Imperative-style Coding with Graph-based Performance”. In: *SysML* (2019). URL: <https://proceedings.mlsys.org/book/272.pdf>.
- [166] Anna Montoya, inversion, Kirill Odintsov, and Martin Kotek. *Home Credit Default Risk*. 2018. URL: <https://kaggle.com/competitions/home-credit-default-risk>.
- [167] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. “Ray: A Distributed Framework for Emerging AI Applications”. In: *OSDI*. 2018, pp. 561–577. DOI: [10.48550/arXiv.1712.05889](https://doi.org/10.48550/arXiv.1712.05889).
- [168] Ingo Müller, Cornelius Ratsch, and Franz Färber. “Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems”. In: *EDBT*. 2014. URL: <https://api.semanticscholar.org/CorpusID:8114547>.
- [169] Carlos Enrique Muñiz-Cuza, Matthias Boehm, and Torben Bach Pedersen. *CAMEO: Autocorrelation-Preserving Line Simplification for Lossy Time Series Compression*. 2025. arXiv: 2501.14432 [cs.DB]. URL: <https://arxiv.org/abs/2501.14432>.
- [170] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *BTW*. 2015, pp. 383–402. URL: <https://dl.gi.de/20.500.12116/2418>.
- [171] Jianmo Ni. *Amazon Product Data - Books*. https://cseweb.ucsd.edu/~jmcauley/datasets/amazon_v2/. 2018.
- [172] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. “LINVIEW: incremental view maintenance for complex analytical queries”. In: *SIGMOD*. 2014, pp. 253–264.
- [173] Adel Nouredine. “PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools”. In: *IE2022*. 2022.
- [174] NVIDIA. *cuBLAS, Basic Linear Algebra on NVIDIA GPUs*. <https://developer.nvidia.com/cublas>. 2024.
- [175] NVIDIA. *CUDA Sparse Matrix lib*. <https://docs.nvidia.com/cuda/cusparsel/>. 2024.

- [176] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture - UNPRECEDENTED ACCELERATION AT EVERY SCALE*. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>. 2020.
- [177] Dan Olteanu. “The relational data borg is learning”. In: *PVLDB*. Vol. 13. 12. 2020, pp. 3502–3515. DOI: 10.14778/3415478.3415572. URL: <https://doi.org/10.14778/3415478.3415572>.
- [178] Kunle Olukotun. “Keynote: “Let the Data Flow!”” In: *CIDR*. 2021.
- [179] OpenBLAS. *OpenBLAS, an Optimized BLAS Library*. <http://www.openblas.net/>. 2024.
- [180] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy G. Mattson. “The TileDB Array Data Storage Manager”. In: *PVLDB* 10.4 (2016), pp. 349–360. DOI: 10.14778/3025111.3025117. URL: <https://doi.org/10.14778/3025111.3025117>.
- [181] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. “BlinkML: Efficient Maximum Likelihood Estimation with Probabilistic Guarantees”. In: *SIGMOD*. 2019, pp. 1135–1152. DOI: 10.1145/3299869.3300077. URL: <https://doi.org/10.1145/3299869.3300077>.
- [182] Ismail Parsa. *KDD Cup 1998 Data*. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5401H>. 1998.
- [183] Arnab K. Paul, Jong Youl Choi, Ahmad Maroof Karimi, and Feiyi Wang. “Machine Learning Assisted HPC Workload Trace Generation for Leadership Scale Storage Systems”. In: *HPDC*. 2022. ISBN: 9781450391993. DOI: 10.1145/3502181.3531457. URL: <https://doi.org/10.1145/3502181.3531457>.
- [184] Karl Pearson. “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. 11. 1901, pp. 559–572. DOI: 10.1080/14786440109462720. URL: <https://doi.org/10.1080/14786440109462720>.
- [185] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al. “Scikit-learn: Machine Learning in Python”. In: *JMLR* 12 (2011), pp. 2825–2830.
- [186] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. “Scikit-learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.
- [187] Tuomas Pelkonen et al. “Gorilla: a fast, scalable, in-memory time series database”. In: *PVLDB*. Vol. 8. 12. 2015, pp. 1816–1827. DOI: 10.14778/2824032.2824078. URL: <https://doi.org/10.14778/2824032.2824078>.
- [188] Arnab Phani and Matthias Boehm. “MEMPHIS: Holistic Lineage-based Reuse and Memory Management for Multi-backend ML Systems”. In: *EDBT*. 2025. URL: <https://phaniarnab.github.io/assets/papers/cikm2022.pdf>.
- [189] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. “UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads”. In: *PVLDB*. Vol. 15. 11. 2022. URL: <https://doi.org/10.14778/3551793.3551842>.
- [190] Arnab Phani, Benjamin Rath, and Matthias Boehm. “LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems”. In: *SIGMOD*. 2021, pp. 1426–1439. URL: <https://doi.org/10.1145/3448016.3452788>.
- [191] Mercedes Piedra et al. *Santander Customer Transaction Prediction*. 2019. URL: <https://kaggle.com/competitions/santander-customer-transaction-prediction>.

- [192] Holger Pirk, Eleni Petraki, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. “Database cracking: fancy scan, not poor man’s sort!” In: *DaMoN@SIGMOD*. 2014, 4:1–4:8. DOI: [10.1145/2619228.2619232](https://doi.org/10.1145/2619228.2619232). URL: <https://doi.org/10.1145/2619228.2619232>.
- [193] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *PVLDB* 6.11 (2013), pp. 1080–1091. DOI: [10.14778/2536222.2536233](https://doi.org/10.14778/2536222.2536233). URL: <https://doi.org/10.14778/2536222.2536233>.
- [194] Vijayshankar Raman and Garret Swart. “How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations”. In: *VLDB*. 2006, pp. 858–869. URL: <http://dl.acm.org/citation.cfm?id=1164201>.
- [195] P. Ratanaworabhan et al. “Fast lossless compression of scientific floating-point data”. In: *DCC*. 2006, pp. 133–142. DOI: [10.1109/DCC.2006.35](https://doi.org/10.1109/DCC.2006.35).
- [196] Alexander Ratner et al. “Snorkel: rapid training data creation with weak supervision”. In: *PVLDB*. Vol. 11. 3. 2017. DOI: [10.14778/3157794.3157797](https://doi.org/10.14778/3157794.3157797). URL: <https://doi.org/10.14778/3157794.3157797>.
- [197] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. “Data programming: Creating large training sets, quickly”. In: *NeurIPS*. Vol. 29. 2016. URL: https://proceedings.neurips.cc/paper_files/paper/2016/file/6709e8d64a5f47269ed5cea9f625f7ab-Paper.pdf.
- [198] Walter Reade. *Categorical Feature Encoding Challenge*. 2019. URL: <https://kaggle.com/competitions/cat-in-the-dat>.
- [199] Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. “Combiner: Full Attention Transformer with Sparse Computation Cost”. In: *NeurIPS*. 2021. URL: https://proceedings.neurips.cc/paper_files/paper/2021/file/bd4a6d0563e0604510989eb8f9ff71f5-Paper.pdf.
- [200] Mark A. Roth and Scott J. Van Horn. “Database compression”. In: *SIGMOD Rec.* 22.3 (1993), pp. 31–39. ISSN: 0163-5808. DOI: [10.1145/163090.163096](https://doi.org/10.1145/163090.163096). URL: <https://doi.org/10.1145/163090.163096>.
- [201] Yousef Saad. “SPARSKIT: a basic tool kit for sparse matrix computations - Version 2”. In: 1990. URL: <https://api.semanticscholar.org/CorpusID:207974787>.
- [202] Brennan Saeta. *Training Performance A user’s guide to converge faster, TF Dev Summit*. 2018.
- [203] Ricardo Salazar, Felix Neutatz, and Ziawasch Abedjan. “Automated Feature Engineering for Algorithmic Fairness”. In: *PVLDB* 14.9 (2021), pp. 1694–1702. DOI: [10.14778/3461535.3463474](https://doi.org/10.14778/3461535.3463474).
- [204] SAP. *SAP HANA Performance Guide for Developers*. https://help.sap.com/doc/05b8cb60dfd94c82b86828ee77f7e0d9/2.0.04/en-US/SAP_HANA_Performance_Developer_Guide_en.pdf. 2019.
- [205] Sebastian Schelter. “Amnesia” - Machine Learning Models That Can Forget User Data Very Fast”. In: *CIDR*. 2020.
- [206] Sebastian Schelter et al. “Automating Large-Scale Data Quality Verification”. In: *PVLDB* 11.12 (2018), pp. 1781–1794. DOI: [10.14778/3229863.3229867](https://doi.org/10.14778/3229863.3229867).
- [207] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. “Learning Linear Regression Models over Factorized Joins”. In: *SIGMOD*. 2016, pp. 3–18. DOI: [10.1145/2882903.2882939](https://doi.org/10.1145/2882903.2882939). URL: <https://doi.org/10.1145/2882903.2882939>.

- [208] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. “Hidden Technical Debt in Machine Learning Systems”. In: *NeurIPS*. 2015, pp. 2503–2511.
- [209] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. “1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs”. In: *INTERSPEECH*. 2014, pp. 1058–1062. URL: https://www.isca-archive.org/interspeech_2014/seide14_interspeech.html.
- [210] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulmaga, et al. “Clay: Fine-Grained Adaptive Partitioning for General Database Schemas”. In: *PVLDB*. Vol. 10. 4. 2016. DOI: 10.14778/3025111.3025125. URL: <http://www.vldb.org/pvldb/vol10/p445-serafini.pdf>.
- [211] C. E. Shannon. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.
- [212] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. “SAGA: A Scalable Framework for Optimizing Data Cleaning Pipelines for Machine Learning Applications”. In: *PACMOD* 1.3 (2023). DOI: 10.1145/3617338. URL: <https://doi.org/10.1145/3617338>.
- [213] Alexander J. Smola and Shriram M. Narayanamurthy. “An Architecture for Parallel Topic Models”. In: *PVLDB* 3.1 (2010), pp. 703–710. DOI: 10.14778/1920841.1920931.
- [214] Rhey T. Snodgrass and Victor F. Camp A.I.R.E. *Radio Receiving For Beginners*. 1922. URL: <https://www.worldradiohistory.com/Archive-Early-Radio-Assorted/Radio-Receiving-for-Beginners-Snodgrass-1922.pdf>.
- [215] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. “MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions”. In: *SIGMOD*. 2019. ISBN: 9781450356435. DOI: 10.1145/3299869.3319854. URL: <https://doi.org/10.1145/3299869.3319854>.
- [216] Michael Stonebraker, Paul Brown, et al. “The architecture of SciDB”. In: *SSDBM*. 2011, pp. 1–16. ISBN: 9783642223501. URL: <https://dl.acm.org/doi/abs/10.5555/2032397.2032399>.
- [217] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. “The Architecture of SciDB”. In: *SSDBM*. 2011, pp. 1–16. DOI: 10.1007/978-3-642-22351-8_1. URL: https://doi.org/10.1007/978-3-642-22351-8_1.
- [218] Mike Stonebraker et al. “C-store: a column-oriented DBMS”. In: *PVLDB*. 2005, pp. 553–564. ISBN: 1595931546. URL: <https://dl.acm.org/doi/10.5555/1083592.1083658>.
- [219] Julia Stoyanovich, Serge Abiteboul, Bill Howe, H. V. Jagadish, and Sebastian Schelter. “Responsible data management”. In: *Commun. ACM* 65.6 (2022). ISSN: 0001-0782. DOI: 10.1145/3488717. URL: <https://doi.org/10.1145/3488717>.
- [220] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische Mathematik* 13.4 (1969), pp. 354–356.
- [221] Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth O. Stanley, and Jeffrey Clune. “Generative Teaching Networks: Accelerating Neural Architecture Search by Learning to Generate Synthetic Training Data”. In: *ICML*. 2020, pp. 9206–9216. URL: <https://doi.org/10.48550/arXiv.1912.07768>.
- [222] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. “ArnetMiner: Extraction and Mining of Academic Social Networks”. In: *KDD*. 2008.
- [223] Teradata. *Teradata Documentation - Table Statements, USING FAST MODE*. https://docs.teradata.com/r/76g1CuvvQ1YBjb2WPIuk3g/Ls7re6GN7m3Tw_faIYIN2Q. 2021.

- [224] The pandas dev team. *Pandas 2.1*. 2010. DOI: [10.5281/zenodo.3509134](https://doi.org/10.5281/zenodo.3509134). URL: <https://github.com/pandas-dev/pandas>.
- [225] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. “Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms”. In: *SIGKDD*. 2013, pp. 847–855.
- [226] Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. “Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse”. In: *ACM Trans. Database Syst.* 41.4 (2016), 21:1–21:38.
- [227] Yuanyuan Tian, Shirish Tatikonda, and Berthold Reinwald. “Scalable and Numerically Stable Descriptive Statistics in SystemML”. In: *Proceedings - International Conference on Data Engineering* (Apr. 2012), pp. 1351–1359. DOI: [10.1109/ICDE.2012.12](https://doi.org/10.1109/ICDE.2012.12).
- [228] Alessandro Ticchi et al. *G-Research Crypto Forecasting*. 2021. URL: <https://kaggle.com/competitions/g-research-crypto-forecasting>.
- [229] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. “Robustness May Be at Odds with Accuracy”. In: *ICLR*. 2019. URL: <https://openreview.net/forum?id=SyxAb30cY7>.
- [230] Ramakrishna Varadarajan, Bibek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. “DBDesigner: A customizable physical design tool for Vertica Analytic Database”. In: 2014, pp. 1084–1095. DOI: [10.1109/ICDE.2014.6816725](https://doi.org/10.1109/ICDE.2014.6816725). URL: <https://doi.org/10.1109/ICDE.2014.6816725>.
- [231] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, et al. “DBDesigner: A customizable physical design tool for Vertica Analytic Database”. In: *ICDE*. 2014, pp. 1084–1095. DOI: [10.1109/ICDE.2014.6816725](https://doi.org/10.1109/ICDE.2014.6816725).
- [232] Paroma Varma and Christopher Ré. “Snuba: automating weak supervision to label training data”. In: *PVLDB*. 3. 2018. DOI: [10.14778/3291264.3291268](https://doi.org/10.14778/3291264.3291268). URL: <https://doi.org/10.14778/3291264.3291268>.
- [233] R. Vink and C. Peters. *Polars: DataFrames for the new era*. 2020. URL: <https://pola.rs/>.
- [234] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. “Mosaic: A Budget-Conscious Storage Engine for Relational Database Systems”. In: *PVLDB* 13.11 (2020), pp. 2662–2675. DOI: [10.14778/3565838.3565858](https://doi.org/10.14778/3565838.3565858). URL: <https://doi.org/10.14778/3565838.3565858>.
- [235] Ashish Vulimiri, Carlo Curino, Brighten Godfrey, Konstantinos Karanasos, and George Varghese. “WANalytics: Analytics for a Geo-Distributed Data-Intensive World”. In: *CIDR*. 2015.
- [236] Maggie Demkin Walter Reade. *Categorical Feature Encoding Challenge II*. 2019. URL: <https://kaggle.com/competitions/cat-in-the-dat-ii>.
- [237] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. “Training Deep Neural Networks with 8-bit Floating Point Numbers”. In: *NeurIPS*. 2018, pp. 7686–7695. URL: <https://proceedings.neurips.cc/paper/2018/hash/335d3d1cd7ef05ec77714a215134914c-Abstract.html>.
- [238] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Ce Zhang, and Onur Mutlu. “Accelerating Generalized Linear Models with MLWeaving: A One-Size-Fits-All System for Any-precision Learning”. In: *PVLDB* 12.7 (2019), pp. 807–821. DOI: [10.14778/3317315.3317322](https://doi.org/10.14778/3317315.3317322). URL: <https://doi.org/10.14778/3317315.3317322>.
- [239] Welch. “A Technique for High-Performance Data Compression”. In: *Computer* 17.6 (1984), pp. 8–19. DOI: [10.1109/MC.1984.1659158](https://doi.org/10.1109/MC.1984.1659158).

- [240] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. “The implementation and performance of compressed databases”. In: *SIGMOD*. Vol. 29. 3. 2000, pp. 55–67. DOI: [10.1145/362084.362137](https://doi.org/10.1145/362084.362137). URL: <https://doi.org/10.1145/362084.362137>.
- [241] Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. “Compact and Computationally Efficient Representation of Deep Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 31.3 (2020), pp. 772–785. DOI: [10.1109/TNNLS.2019.2910073](https://doi.org/10.1109/TNNLS.2019.2910073).
- [242] Lucas Wilkinson, Kazem Cheshmi, and Maryam Mehri Dehnavi. “Register Tiling for Unstructured Sparsity in Neural Network Inference”. In: *PLDI* 7 (2023). DOI: [10.1145/3591302](https://doi.org/10.1145/3591302). URL: <https://doi.org/10.1145/3591302>.
- [243] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. “SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units”. In: *PVLDB* 2.1 (2009), pp. 385–394. DOI: [10.14778/1687627.1687671](https://doi.org/10.14778/1687627.1687671). URL: <https://doi.org/10.14778/1687627.1687671>.
- [244] Ian H. Witten, Radford M. Neal, and John G. Cleary. “Arithmetic coding for data compression”. In: *Commun. ACM* 30.6 (1987), pp. 520–540. ISSN: 0001-0782. DOI: [10.1145/214762.214771](https://doi.org/10.1145/214762.214771). URL: <https://doi.org/10.1145/214762.214771>.
- [245] Stephen Wolfram. *A New Kind of Science - Data Compression*. Wolfram-Media, 2002, p. 1069. URL: <https://www.wolframscience.com/nks/notes-10-5--history-of-data-compression/>.
- [246] Qiang Wu and Ding-Xuan Zhou. “SVM Soft Margin Classifiers: Linear Programming versus Quadratic Programming”. In: *Neural Comput.* 17.5 (2005), pp. 1160–1187. ISSN: 0899-7667. DOI: [10.1162/0899766053491896](https://doi.org/10.1162/0899766053491896). URL: <https://doi.org/10.1162/0899766053491896>.
- [247] Yuncheng Wu, Shaofeng Cai, Xiaokui Xiao, Gang Chen, and Beng Chin Ooi. “Privacy Preserving Vertical Federated Learning for Tree-based Models”. In: *PVLDB* 13.11 (2020), pp. 2090–2103.
- [248] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. “Helix: Holistic Optimization for Accelerating Iterative Machine Learning”. In: *PVLDB* 12.4 (2018), pp. 446–460.
- [249] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. “Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities”. In: *SIGMOD*. 2021, pp. 2639–2652. DOI: [10.1145/3448016.3457566](https://doi.org/10.1145/3448016.3457566). URL: <https://doi.org/10.1145/3448016.3457566>.
- [250] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. “Online Deduplication for Databases”. In: *SIGMOD*. 2017, pp. 1355–1368. DOI: [10.1145/3035918.3035938](https://doi.org/10.1145/3035918.3035938). URL: <https://doi.org/10.1145/3035918.3035938>.
- [251] Huanrui Yang, Wei Wen, and Hai Li. “DeepHoyer: Learning Sparser Neural Network with Differentiable Scale-Invariant Sparsity Measures”. In: *ICLR*. 2020. URL: <https://openreview.net/forum?id=rylBK34FDS>.
- [252] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. “Federated Machine Learning: Concept and Applications”. In: *ACM Trans. Intell. Syst. Technol.* 10.2 (2019), 12:1–12:19.
- [253] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. “Qd-tree: Learning Data Layouts for Big Data Analytics”. In: *SIGMOD*. 2020, pp. 193–208. DOI: [10.1145/3318464.3389770](https://doi.org/10.1145/3318464.3389770). URL: <https://doi.org/10.1145/3318464.3389770>.

- [254] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, et al. “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”. In: *NSDI*. 2012, p. 2. URL: <https://dl.acm.org/doi/10.5555/2228298.2228301>.
- [255] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *NSDI*. 2012, pp. 15–28.
- [256] Ce Zhang, Arun Kumar, and Christopher Ré. “Materialization optimizations for feature selection workloads”. In: *SIGMOD*. 2014, pp. 265–276.
- [257] Dongqing Zhang et al. “LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks”. In: *ECCV*. 2018. arXiv: [1807.10029](https://arxiv.org/abs/1807.10029).
- [258] Hantian Zhang et al. “ZipML: training linear models with end-to-end low precision, and a little bit of deep learning”. In: *ICML*. 2017, pp. 4035–4043. URL: <https://dl.acm.org/doi/10.5555/3305890.3306098>.
- [259] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. “ZipML: Training Linear Models with End-to-End Low Precision, and a Little Bit of Deep Learning”. In: *ICML*. 2017, pp. 4035–4043. URL: <https://proceedings.mlr.press/v70/zhang17e.html>.
- [260] Jiuqing Zhang et al. “High-Ratio Compression for Machine-Generated Data”. In: *PACMMOD*. Vol. 1. 4. 2023. DOI: [10.1145/3626732](https://doi.org/10.1145/3626732). URL: <https://doi.org/10.1145/3626732>.
- [261] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. “Trained Ternary Quantization”. In: *ICLR*. 2017. URL: https://openreview.net/forum?id=S1%5C_pAu9xl.
- [262] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Trans. Inf. Theory* 23.3 (1977), pp. 337–343. DOI: [10.1109/TIT.1977.1055714](https://doi.org/10.1109/TIT.1977.1055714). URL: <https://doi.org/10.1109/TIT.1977.1055714>.
- [263] Jacob Ziv and Abraham Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Trans. Inf. Theory* 24.5 (1978), pp. 530–536. DOI: [10.1109/TIT.1978.1055934](https://doi.org/10.1109/TIT.1978.1055934). URL: <https://doi.org/10.1109/TIT.1978.1055934>.
- [264] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. “Super-Scalar RAM-CPU Cache Compression”. In: *ICDE*. 2006, p. 59. DOI: [10.1109/ICDE.2006.150](https://doi.org/10.1109/ICDE.2006.150). URL: <https://doi.org/10.1109/ICDE.2006.150>.



A.1 Arithmetic Encoding

If we have three distinct values: x , y , and z , each with an equal probability of $0.\bar{3}$, then an optimal encoding, according to Shannon theorem [211], would use 1.58 bits per value, which is close to possible with arithmetic coding. Arithmetic encoding compresses data by using the frequencies of distinct values (plus an 'end' value) associating with a relatively sized fraction in the range $[0, 1]$. e.g., if the frequency is equal for x , y and z , $x = [0, 0.3)$, $y = [0.3, 0.6)$, $z = [0.6, 0.9)$ and $end = [0.9, 1]$. Encoding the sequence " $x x y z end$ " recursively applies the fractional range as shown in A.1.

$$\begin{aligned}
 x &\rightarrow [0, 0.3) \\
 x &\rightarrow [0, 0.09) &= [0, 0.3 \cdot 0.3) \\
 y &\rightarrow [0.027, 0.054) &= [0. + 0.09 \cdot 0.3, 0.09 \cdot 0.6) \\
 z &\rightarrow [0.0432, 0.0513) &= [0.027 + (0.054 - 0.027) \cdot 0.6, 0.054 - (0.054 - 0.027) \cdot (1 - 0.9)) \\
 end &\rightarrow [0.05049, 0.0513) &= [0.0432 + (0.0513 - 0.0432) \cdot 0.9, 0.0513)
 \end{aligned} \tag{A.1}$$

Any value in the range encodes the sequence, and we can decode the value via recursive lookups and modification of intermediate ranges. For instance, choosing 0.051 as the encoded value decompress as shown in A.2.

$$\begin{aligned}
 0.051 &\in [0, 0.3) &\rightarrow 0.051/0.3 &= 0.16\bar{9} &\rightarrow x \\
 0.16\bar{9} &\in [0, 0.3) &\rightarrow 0.16\bar{9}/0.3 &= 0.5\bar{6} &\rightarrow x \\
 0.5\bar{6} &\in [0.3, 0.6) &\rightarrow (0.5\bar{6} - 0.3)/0.3 &= 0.79\bar{6}2 &\rightarrow y \\
 0.8 &\in [0.6, 0.9) &\rightarrow (0.79\bar{6}2 - 0.6)/0.3 &= 0.9\bar{6}2 &\rightarrow z \\
 0.9\bar{6}2 &\in [0.9, 1] &&&\rightarrow end
 \end{aligned} \tag{A.2}$$

A.

The following is a basic example of Arithmetic encoding. Note that for it to work on longer sequences, the floating points must use infinite precision (which is not the case in the example).

```
import random

a = [0, 0.3, 0.6, 0.9, 1] # Probability distribution
tokens = [0,0,1,2,3] # Values to compress

range = [0.0, 1.0] # Valid compression range
for e in tokens: # encode
    diff = range[1] - range[0]
    range = [range[0] + diff * a[e],
            range[1] - diff * (1 - a[e+1])]

# compressed value
r = random.random() * (range[1] - range[0]) + range[0]

while r < a[-2]: # decode
    id = 0
    while a[id] < r: id += 1
    # min-max normalization
    r = (r - a[id-1]) / (a[id] - a[id-1])
    print(id-1, end=" ")
```

A.2 Sparse Operations Analysis

To properly analyze the operational cost of sparse linear algebra operations past big-O notations, we need to split the cost up into different elements.

1. Operation execution: How much does a single operation cost?
2. Index lookups: How much indirection via index calculation?
3. Cache efficiency: How many bytes do we load per operation?

This cost-breakdown is a simplified view since it is missing parallelization, blocking, and multi-level cache hierarchy are influential factors for accurate estimates.

Sum

Starting with a sum operation on a matrix, a dense row or column major implementation would look something like

```
public static double sum(double[] a){
    double r = 0; // O(1)
    for(int i = 0; i < a.length; i++) // O(A)
        r += a[i];
    return r;
}
```

Dense Sum: The $\mathcal{O}(\mathbf{A}) = \mathcal{O}(nm)$ is consistent with the initial table [Table 2.1](#). The Operation cost is once per value via addition into the running sum. There are no index lookups except directly into the row/col major array, and therefore, no indirections. Finally, the cache efficiency is good, by sequentially loading all double values in the array. Each value fills 8 bytes, therefore we load 8 bytes per value processed. It is well known that this operation is memory-bandwidth bound because we perform disproportionally little computation compared to loading values.

Sparse Sum: A sparse COO, CSR, CSC implementation (see [Figure 2.2](#)), could call the same method as the dense, except using only value arrays reducing the asymptotic execution time to $\mathcal{O}(\mathbf{A}_{\neq 0})$. Index lookups and cache efficiency are similar, but the number of values processed is reduced. Similar to dense, this operation is memory-bandwidth bound.

Row/Col Sum

To calculate row or column sums, we need additional information. Assuming a row-major layout and calculating column sums, the dense operation becomes:

```
public static double[] colSum(double[] a, int nCol){
    double[] r = new double[nCol]; // O(m)
    for(int i = 0; i < a.length; i++) // O(A)
        r[i % nCol] += a[i];
    return r;
}
```

Dense Row/Col Sum: To follow the cost decomposition, the operational cost is equivalent to the full sum with the addition. However, for indexing, we have additionally to calculate the output cell via modulo (row sum replace modulo with division), and we now potentially load two double values per operation, one for the input and one for the output. Assuming that the output number of columns is small, the overhead of loading the output cell is small, while loading input cells from a is more expensive because they are not reused. Row sum and column sum are both also memory-bandwidth bound. Therefore, assuming a small output number of columns, the execution time is equivalent to a normal sum.

Unlike the sparse sum, sparse row/col sum can use a trick:

```
public static double[] colSum(int[] cols, double[] a, int nCol){
    double[] r = new double[nCol]; // O(R)
    for(int i = 0; i < a.length; i++) // O(A_nnz)
        r[cols[i]] += a[i];
    return r;
}
```

Sparse Col Sum: Sparse column sum have $\mathcal{O}(\mathbf{R} + \mathbf{A}_{\neq 0})$. The asymptotic \mathbf{R} , is from allocating the output vector. Both COO and CSR can loop through the non-zero value arrays and use the column offset array to index into the right output cells. However, there is an overhead in index lookups, that has an indirection in the column offset array that also offsets into the result array. We have to load a double from the non-zero array, a double from the output, and an integer from the column offset array, making each value processed more expensive on the cache. Therefore, the cache overhead is increased to 20 bytes per value processed. Similarly to dense, we can argue that the result vector can be reused if the output is small and reduce the cache usage to 12 bytes per value processed. With these estimates, we can theoretically say that a sparsity of < 0.6 is required before sparse column sums on CSR or COO should begin to be as fast as dense. Depending on the order of the columns in CSR or COO, the sparse implementation can be very cache-unfriendly via randomly shuffled indexes. Therefore, it is preferred that the indexes are sorted.

```
public static double[] rowSumCSR(int[] rOff, double[] a,
    int nRow){
    double[] r = new double[nRow]; // O(R)
    for(int i = 0; i < rOff.length - 1; i++) // O(n + A_nnz)
        for(int j = rOff[i]; i < rOff[i+1]; j++)
            r[i] += a[j];
    return r;
}
```

Sparse Row Sum: The CSR implementation diverges from the COO implementation when calculating the row sum. However, both implementations follow $\mathcal{O}(\mathbf{R} + n + \mathbf{A}_{\neq 0})$, COO can use the exact same method from column sum, except it should use the row offsets instead of the column offsets for the first input. The changes for CSR make each row's value range is checked in an outer loop, and then in an inner loop, each individual row's values are added to the corresponding output cell. The CSR rowSum implementation is close to equivalent in cache efficiency to dense with a slight overhead in loading two-row offset values per row processed, while it also does not incur any significant index lookups. In practice, assuming some values on each row, the CSR implementation is more efficient, while if the matrix is ultra sparse with many empty rows, COO is better. Based on this analysis, CSR should be close to equivalent in performance to dense even when $\mathbf{A}_{\neq 0}$ approaches \mathbf{A} , with an overhead on many rows, and speeds up linearly with reductions in non-zero values.

Max

Dense Max: The dense max operation have the same behavior as sum, with a slight increase in the operational cost. However, since it is still a memory bandwidth bound operation, it should exhibit similar execution times to sum.

```
public static double max(double[] a){
    double r = a[0]; // O(1)
    for(int i = 1; i < a.length; i++) // O(A)
        r = a[i] > r ? a[i] : r;
    return r;
}
```

Sparse Max: The sparse max, also behaves similarly to the sparse sum, with an small change to the initial state handling if the number of non-zero values contained $A_{\neq 0}$ is equal to the number of cells A .

```
public static double max(double[] a, int nCol, int nRow){
    double r = nCol * nRow > a.length ? 0 : a[0]; // O(1)
    for(int i = 0; i < a.length; i++) // O(A)
        r = a[i] > r ? a[i] : r;
    return r;
}
```

Row/Col Max

This section describe the binary operations Max, which also is applicable to other operations such as Min with minor modifications.

Dense Row/Col Max: Dense row or column max is equivalent to the dense sum (Appendix A.2), with minor modifications. The operational performance is also equivalent.

Calculating the maximum value in COO sparse formats can be done via the following code.

```
public static double[] colMaxCOO(int[] cols, double[] a, int
    nCol,
    int nRow){
    int[] counts = new int[nCol]; // O(R)
    for(int i = 0; i < a.length; i++) // nnz col count O(A_nnz)
        counts[cols[i]]++;
    double[] r = new double[nCol]; // O(R)
    for(int i = 0; i < nCol; i++) // initialize result O(R)
        r[i] = counts[i] == nRow ? Double.MIN_VALUE : 0;
    for(int i = 0; i < a.length; i++) // process O(A_nnz)
        r[cols[i]] = a[i] > r[cols[i]] ? a[i] : r[i];
    return r;
}
```

Sparse Row/Col Max: Because it is unknown which columns contain zero values, the sparse algorithm requires extra processing. The example above uses a two-pass algorithm for COO that first counts non-zeros in columns, followed by an initialization of the result vector. After the preprocessing, we can iterate through the column indexes and values. Overall, this implementation requires a two-pass over the number of non-zero, and additionally, we allocate both an integer array and a double array. In C++, it would be possible to reuse the allocation by, for instance, using long counts and recasting the long array to an output of doubles. The first loop to count is more cache efficient, loading 8 bytes per processed value, while the final loop loads 20.

Calculating the row max on a CSR allows skipping the counting since the row offsets implicitly contain the number of non-zero values of rows. CSR simplifies the algorithm to:

```
public static double[] rowMaxCSR(int[] rOff, int[] cols,
    double[] a, int nCol, int nRow){
    double[] r = new double[nRow]; // O(n)
    for(int i = 0; i < rOff.length - 1; i++){ // O(n + A_nnz)
        int s = rOff[i];
        int e = rOff[i+1];
        r[i] = e - s == nCol ? Double.MIN_VALUE : 0;
        for(int j = s; j < e; j++)
            r[i] = a[j] > r[i] ? a[j] : r[i];
    }
    return r;
}
```

$\mathcal{O}(n + \mathbf{A}_{\neq 0})$ is still the same. However, the CSR implementation only requires a single pass over the data. We still note that the number of non-zeros can be vastly bigger than the number of rows.

Sparse Unsafe ($\mathbf{A} + \mathbf{M}$)

The sparse unsafe operations typically return dense outputs. Therefore, the asymptotic runtime is dictated by the output dimension $\mathcal{O}(\mathbf{A})$.

Binary Sparse Safe ($\mathbf{A} \cdot \mathbf{M}$)

Sparse safe operations on the other hand can be optimized. If the other input is a scalar, vector, or matrix, does not change the asymptotic runtime but it does lower the cache efficiency.

```
public static double[] mult(double[] a, double[] b){
    double[] ret = new double[a.length]; // O(A)
    for(int i = 0; i < a.length; i++) // O(A)
        ret[i] = a[i] * b[i];
    return ret;
}
```

Dense: The dense example above assume both input matrices are linearized similarly (row or column). If this is true, it loops through the inputs and assign a new output. The operation is memory bandwidth bound, and loads 24 bytes per iteration of the for loop.

```
public static COO multCOO(int[] rows, int[] cols,
    double[] a, double[] b, int nCol){
    double[] ret = new double[a.length]; // O(A_nnz)
    int nnz = 0;
    for(int i = 0; i < a.length; i++) // O(A_nnz)
        nnz += (ret[i] = a[i] * b[rows[i] * nCol + cols[i]])
            == 0 ? 0 : 1;
    if(nnz < a.length) // compact
        return compactCOO(rows, cols, ret, nnz); // O(A_nnz)
    return new COO(rows, cols, ret);
}
```

Sparse COO: Assuming the input matrix is row-major linearized into \mathbf{b} , and the row and column indexes are immutable for COO. Then the COO execution time can scale in number of non zeros $\mathcal{O}(\mathbf{A}_{\neq 0})$, as shown in the above code. The COO version loads 8 more bytes per value processed in the inner loop, making it 32 bytes per value processed. Since it is possible for \mathbf{b} to contain zero values, the resulting COO can be postprocessed ($\mathcal{O}(\mathbf{A}_{\neq 0})$) in a second pass if the output contains less non-zero values.

```

public static CSR multCSR(int[] rOff, int[] cols,
    double[] a, double[] b, int nCol){
    double[] ret = new double[a.length]; // O(A_nnz)
    int nnz = 0;
    for(int i = 0; i < rOff.length - 1; i++){ // O(n + A_nnz)
        int s = rOff[i];
        int e = rOff[i+1];
        int off = i * nCol; // precompute row offset
        for(int j = s; j < e; j++)
            ret[j] = a[j] * b[off + cols[j]];
    }
    if(nnz < a.length) // compact
        return compactCSR(rOff, cols, ret, nnz); // O(n + A_nnz)
    return new CSR(rOff, cols, ret);
}

```

Sparse CSR: The sparse CSR comes with two advantages. First, the inner loop processing reduce the number of loaded bytes by 4, to 28 per value processed. Second, that we guarantee accessing b in a consistent pattern for each row. However, CSR has a disadvantage if many rows are empty, because it forces iteration though the row offsets. Similar to COO, CSR can postprocess the output if the number of non-zero values is reduced.

Matrix Multiplication (A@M)

This section describe the matrix multiplication baselines in dense and sparse versions. It is a simplification of actual implementations that further optimize via cache blocking of inputs, and vectorized kernels for efficient processing of ranges.

```

public static double[] mm(double[] a, double[] b,
    int rowA, int colB){
    double [] ret = new double[rowA * colB]; // O(nk)
    int cd = a.length / rowA; // common dimension
    for(int i = 0; i < rowA; i++){ // O(nmk)
        int off0 = i * colB;
        int offA = i * cd;
        for(int k = 0; k < cd; k++, offA++){
            int offB = k * colB;
            double av = a[offA];
            for(int j = off0; j < off0 + colB; j++, offB++)
                ret[j] += av * b[offB];
        }
    }
    return ret;
}

```

Dense: A simplified dense matrix multiplication without vectorization or cache blocking of two row-major inputs could be implemented, as shown above. Notably, each multiplication performed in the inner loop loads 16 bytes. However, unlike the previous operations, matrix multiply is not memory bandwidth bound. Instead, it is compute bound because many of the values loaded are reused for multiple computations [94, 29]. The cache reuse potential is optimal when multiplying small square blocks that, together with the program, fit in the lowest level of cache [94].

```

public static double[] mmCOO(int[] rows, int[] cols, double[] a,
    double[] b, int rowA, int colB){
    double [] ret = new double[rowA * colB]; // O(nk)
    for(int i = 0; i < a.length; i++){ // O(A_nnz k)
        double aV = a[i];
        int off0 = rows[i] * colB;
        int offB = cols[i] * colB;
        for(int j = off0; j < off0 + colB; j++, offB++){
            ret[j] += aV * b[offB];
        }
    }
    return ret;
}

```

Sparse COO: COO sparse matrix multiplication with a dense matrix asymptotically costs $\mathcal{O}(nk + \sum_{i=1}^n \mathbf{A}_{i,\neq 0}k)$. Where nk is the allocation of the output, that could dominate in ultra sparse cases, and the matrix multiplication itself scaling in nnz cells. Assuming the coordinates are sorted row-major, and the output similarly is row-major, the cache efficiency is decent, with the outer loop loading 16 bytes per non zero value. The inner loop does not load many values with 16 bytes per loop iteration, of which both the result cell and the dense input cell could be reused from cache often. Therefore, if the input is sparse COO based matrix multiplication should be faster.

```

public static double[] mmCSR(int[] rowOff, int[] cols,
    double[] a, double[] b, int rowA, int colB){
    double [] ret = new double[rowA * colB]; // O(nk)
    for(int i = 0; i < rowOff.length-1; i++){ // O(A_nnz k + n)
        int off0 = i * colB;
        for(int j = rowOff[i]; j < rowOff[i+1]; j++){
            double aV = a[j];
            int offB = cols[j] * colB;
            for(int k = off0; k < off0 + colB; k++, offB++){
                ret[k] += aV * b[offB];
            }
        }
    }
    return ret;
}

```

Sparse CSR: The CSR right matrix multiplication adds a the extra loop, similar to other operations, that adds an asymptotic cost for each row processed. However, it does allow reducing the offset calculation from every cell, to once per row. The rest of the behavior is similar to COO. Therefore, CSR is slightly better once a non trivial number of non-zeros are contained in the input.

Other Input Sparse: If both input matrices are sparse, then the innermost for loop can be optimized to only process non-zero elements of the individual rows. Further reducing execution time. To support this behavior with COO, it is ideal if the cells are sorted row-major, to allow a binary search to find row boundaries. If CSR, the row offsets, suffice to find non-zero values making it constant time lookups.

Sparse Outputs: In most cases the matrix multiplication produce a dense output with many non-zero values. However, some cases where the input matrices are ultra sparse, and contain full zero rows on the left matrix or columns on the right, produce sparse results. Therefore, specializations for this case is also beneficial.

Transpose Self Matrix Multiply ($A^t @ A$)

The final example operation is transpose self matrix multiplication (TSMM) that, without materializing the transposed version of itself does the multiplication.

```
public static double[] tsmm(double[] a, int nRow, int nCol) {
    double[] ret = new double[nCol * nCol]; //  $O(m^2)$ 
    for(int i = 0; i < nCol; i++) { //  $O(nm^2)$ 
        int off0 = i * nCol;
        for(int j = 0; j < nRow; j++) {
            int offB = j * nCol;
            double aV = a[offB + i];
            // skip based on column processed
            offB = offB + i;
            for(int k = off0 + i; k < off0 + nCol; k++, offB++)
                ret[k] += aV * a[offB];
        }
    }
    copyUpperHalf(ret, nCol); //  $O(m^2)$ 
    return ret;
}
```

Dense: The above code shows a version of dense tsmm. The above implementation does not apply cache blocking, which is critical for improved performance. However, the implementation does show the correct asymptotic behavior. Importantly, for TSMM, we can suffice with calculating the top half of the result and copy it to the bottom half because the result is a symmetric matrix.

```
public static double[] tsmmCSR(int[] rowOff, int[] cols,
    double[] a, int nCol) {
    final double[] ret = new double[nCol * nCol]; //  $O(m^2)$ 
    for(int i = 0; i < rowOff.length - 1; i++) { //  $O(\dots + n)$ 
        final int s = rowOff[i];
        for(int j = s; j < rowOff[i + 1]; j++) { //  $O(\text{sum}(r\_nnz^2))$ 
            final double aV = a[j];
            final int col = cols[j] * nCol;
            // Start from j to only calculate upper half
            for(int k = j; k < e; k++) { //  $O(\text{sum}(r\_nnz))$ 
                ret[cols[k] + col] += aV * a[k];
            }
        }
    }
    copyUpperHalf(ret, nCol);
    return ret;
}
```

Sparse: Above is a sparse CSR-based transpose self-multiplication. Similar to the dense version, the sparse TSMM calculates only half of the output (seen in the last inner for loop) and copies the result to the other half. In ultra-sparse cases, the output allocation of $\mathcal{O}(m^2)$ dominates the execution time. However, more commonly it is the number of non-zeros $\mathcal{O}(\sum_{i=1}^n (\mathbf{A}_{i,\neq 0})^2)$. However, if n is large, m is small, and the number of non zero values is low, then the number of rows become the dominating factor for CSR $\mathcal{O}(n)$, while, COO does not suffer from this constant factor.

A.3 CSR-VI Operations Analysis

Vector Multiplication

While the asymptotic runtime does not improve for vector multiplication in CSR-VI, it does improve the cache efficiency if the number of distinct values is low. Vector multiplications are memory bandwidth bound typically. The following code is the CSR baseline for vector multiplication:

```
public static double[] mvCSR(int[] rowOff, int[] cols,
    double[] a, double[] b, int rowA){
    double [] ret = new double[rowA]; // O(nk)
    for(int i = 0; i < rowOff.length-1; i++){ // O(A_nnz + n)
        double t = 0.0;
        for(int j = rowOff[i]; j < rowOff[i+1]; j++){
            t += a[j] * b[cols[j]];
            ret[i] += t;
        }
    }
    return ret;
}
```

CSR baseline: We can see the inner loop have to load two doubles, one from the csr value array a and one from b . Furthermore, we need to load the column index, to indicate which value in b to multiply with. The ret cell is reused for the entire inner loop, therefore cached, while the other two are respectively accessed in sequence (on a) and somewhat at random (on b). In total, an estimate is we load 20 bytes per iteration of the inner loop.

```
public static double[] mvCSRVI(int[] rowOff, int[] cols,
    int[] m, double[] d, double[] b, int rowA){
    double [] ret = new double[rowA]; // O(nk)
    for(int i = 0; i < rowOff.length-1; i++){ // O(A_nnz + n)
        double t = 0.0;
        for(int j = rowOff[i]; j < rowOff[i+1]; j++){
            t += d[m[j]] * b[cols[j]];
            ret[i] += t;
        }
    }
    return ret;
}
```

CSR-VI: The CSR-VI implementation adds an indirection in the inner loop that via a mapping, m , looks up the distinct values in d . If the number of distinct values is small, we can assume that the entire d is loaded in cache, thereby reducing the loaded bytes to 16 per inner iteration in the example. If the mapping is allocated in smaller formats with $\#B < 32$ as in the example the loaded values can be further reduced. For instance with two distinct values to 12bytes and 1 bit per inner iteration. The reduction in loaded values gives performance gains because the operation is memory bandwidth bound.

A.4 Kernel Function Example

Let a and b be points in two-dimensional space ($a, b \in \mathbb{R}^2$), where $a = [a_1, a_2]$ and $b = [b_1, b_2]$. We need to prove the kernel function of the feature map:

$$\varphi(a) = \varphi([a_1, a_2]) = [a_1, a_2, a_1^2 + a_2^2]$$

Is equal to:

$$k(a, b) = a^\top b + \|a\|^2 \|b\|^2$$

The definition of a kernel function is:

$$k(a, b) = \langle \varphi(a), \varphi(b) \rangle$$

Therefore, after substitutions, the function simplifies accordingly:

$$\begin{aligned} k(a, b) &= \langle \varphi(a), \varphi(b) \rangle \\ &= \varphi(a)^\top \varphi(b) \\ &= \varphi([a_1, a_2])^\top \varphi([b_1, b_2]) \\ &= [a_1, a_2, a_1^2 + a_2^2]^\top [b_1, b_2, b_1^2 + b_2^2] \\ &= a_1 b_1 + a_2 b_2 + (a_1^2 + a_2^2)(b_1^2 + b_2^2) \\ &= [a_1, a_2]^\top [b_1, b_2] + (a_1^2 + a_2^2)(b_1^2 + b_2^2) \\ &= a^\top b + (a_1^2 + a_2^2)(b_1^2 + b_2^2) \\ &= a^\top b + \left(\sqrt{a_1^2 + a_2^2} \right)^2 \cdot \left(\sqrt{b_1^2 + b_2^2} \right)^2 \\ &= a^\top b + \|a\|^2 \|b\|^2 \end{aligned}$$