UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads

Arnab Phani
Graz University of Technology
phaniarnab@gmail.com

Lukas Erlbacher
Graz University of Technology
lukas.erlbacher@student.tugraz.at

Matthias Boehm
Graz University of Technology
m.boehm@tugraz.at

ABSTRACT

Data science pipelines are typically exploratory. An integral task of such pipelines are feature transformations, which transform raw data into numerical matrices or tensors for training or scoring. There exist a wide variety of transformations for different data modalities. These feature transformations incur large computational overhead due to expensive string processing and dictionary creation. Existing ML systems address this overhead by static parallelization schemes and interleaving transformations with model training. These approaches show good performance improvements for simple transformations, but struggle to handle different data characteristics (many features/distinct items) and multi-pass transformations. A key observation is that good parallelization strategies for feature transformations depend on data characteristics. In this paper, we introduce UPLIFT, a framework for *P*aralle*LI*zing *F*eature *T*ransformations. UPLIFT constructs a fine-grained task graph for a set of transformations, optimizes the plan according to data characteristics, and executes this plan in a cache-conscious manner. We show that the resulting framework is applicable to a wide range of transformations. Furthermore, we propose the FTBench benchmark with transformations and datasets from various domains. On this benchmark, UPLIFT yields speedups of up to 31.6x (9.27x on average) compared to state-of-the-art ML systems.

PVLDB Reference Format:
Arnab Phani, Lukas Erlbacher, and Matthias Boehm. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. PVLDB, 15(11): 2929 - 2938, 2022.
doi:10.14778/3551793.3551842

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/damslab/reproducibility (in the folder UPLIFT).

1 INTRODUCTION

Machine Learning (ML) is the foundation of many data-driven applications, which increasingly utilize multi-modal data comprised of structured attributes, images, speech, video, graphs, and text [70, 109]. Data scientists apply various techniques for transforming the raw data into a form suitable for model training and scoring. Common transformations include encoding numerical columns using normalization and binning; encoding categorical columns using recoding (dictionary encoding), dummy coding (one-hot encoding), and feature hashing; as well as vector embeddings for text or graphs. These transformations exhibit different runtime and accuracy characteristics. The variety of data and transformations requires practitioners to explore alternative transformations. While prior work has focused on runtime improvements for pre-processing and model training, feature transformations have received much less attention.

Challenges of Feature Transformations: Feature transformations are typically applied in two phases: (1) A *build* phase obtains the necessary metadata such as a dictionary of distinct items, and (2) an *apply* phase uses those dictionaries to transform the data and combine the feature outputs. This multi-pass nature and potentially many or large dictionaries can render simple, data-parallel execution ineffective. Major challenges include (1) a large number of output columns, (2) many distinct items per column (up to millions [114]), (3) sparsity and cardinality skew, (4) expensive string processing (e.g., hashing and parsing), (5) ultra-sparse outputs, (6) larger-than-memory output data (e.g., due to replicated embeddings), and (7) a wide diversity of transformations.

Feature Transformations in ML Systems: Many end-to-end ML systems and libraries—including TensorFlow TFX [11, 12], MLFlow [119], Sagemaker [62], MLlib/spark.ml [67], Scikit-learn [76], ML.NET [6], and SystemDS [14, 15]—provide native support for data pre-processing and feature transformations. These systems address the large overhead of feature transformations with techniques such as caching and reuse [24, 79, 113] to avoid redundant pre-processing operations, interleaving single-pass, element-wise transformations with data loading [72], parallelizing independent column transformations [1], and static data parallelism [6, 67, 72, 76]. However, to the best of our knowledge, no prior work offers a principled approach to efficiently execute feature transformation workloads according to their data characteristics.

UPLIFT Framework Overview: For executing a feature transformation workload efficiently, we aim to exploit the available parallelism and cache-conscious runtime operations. Given a dataset with known characteristics, our UPLIFT framework constructs an execution plan with the goal of minimizing compute time. Inspired by task and operator scheduling in Ray [71] and TensorFlow [3], we construct a fine-grained task-dependency graph for the different phases of all column encoders. A rule-based optimizer then rewrites the task graph according to data, hardware, and operation characteristics, and submits it for concurrent execution. Similar to vectorized query execution [18] and fused operator pipelines [16], we devise cache-conscious runtime techniques, and fine-grained data-parallelism on row partitions, while satisfying memory constraints. Applying these ideas to feature transformation workloads—with an optimizer that exploits the known and estimated properties of these transformations—yields a clean and very effective framework.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551842

Contributions: Our main contributions are the UPLIFT framework for efficient feature transformations, and a dedicated feature transformation benchmark. UPLIFT is fully integrated in Apache SystemDS[1] [15] as a representative ML system with local, distributed, and federated backends. Our technical contributions are:

- *Background:* We survey commonly used feature transformations, existing libraries, user APIs, as well as existing parallelization strategies and optimizations in Section 2.
- *UPLIFT System Architecture:* We then discuss the overall architecture of UPLIFT with different task types, transformations, and different parallelization strategies in Section 3.
- *FTBench Benchmark:* As a basis for our evaluation, we define a new benchmark for feature transformations, FTBench, covering a realistic mix of data characteristics, feature types, and combination of transformations in Section 4.
- *Experiments:* Finally, we report on extensive experiments in Section 5. We show that UPLIFT significantly improves performance compared to baselines such as SystemDS-Base, TensorFlow [3], spark.ml [67], Scikit-learn [76], and others.

2 BACKGROUND AND PRELIMINARIES

In this section, we discuss commonly used feature transformations for various types of data, different implementations, APIs and configurations, as well as existing optimization techniques.

2.1 Feature Transformations

Feature transformation is a part of feature engineering and scales, modifies, or converts features before numerical computations. Feature transformations are often placed after cleaning and either before or within the training part of an ML pipeline. During model training, we transform the entire dataset and obtain its metadata. During scoring, this metadata is then used for consistent batch- or mini-batch transformations. Transforming the entire dataset exhibits more parallelization opportunities, whereas batch-wise transformations allow overlapping data loading, transforming, and training/scoring [72]. In the following, we categorize the transformations based on the data modalities. Table 1 summarizes the following commonly applied transformations for structured data, which are part of most data science pipelines [81, 114]:

Numerical Transformations: These transformations are used on numeric features. Examples include simple aggregations (min, max, mean) or scaling, normalization, and binning (discretization). Normalization brings numeric features to a standard scale via methods such as scale&shift, min/max-scaling, log-scaling, or Z-scoring for well-behaved training. Binning maps continuous features into discrete categories (integers), with fixed-size bin boundaries (equi-width), or equal frequency per bin and thus, variable boundaries (equi-height). Normalization and binning require multiple passes: a first pass (build) computes statistics (e.g. min/max, quantiles), and a second pass (apply) uses those statistics to transform the data. While parallelizing the apply phase is straightforward, a data-parallel build requires parallel collection and merging of the partial statistics. Pass-through encoders forward numerical data with optional conversion (*) of value types (e.g., strings) where needed.

[1]The source code is available at https://github.com/apache/systemds, specifically in the package src/main/java/org/apache/sysds/runtime/transform.

Table 1: Common Multi-pass Transformations.

Transformation	Build Input	Build Output	Apply Output
Recoding	Nominal	Dictionaries	Integer
Feature Hashing	Nominal	None	Integer
Binning	Numeric*	Bin boundaries	Integer
Pass-through	Numeric*	None	Numeric
Dummy-coding	Integer	Offsets	Sparse vectors

Categorical Transformations: Common techniques for encoding categories are recoding, dummy-coding (one-hot encoding), and feature hashing. Recoding and feature hashing transform a categorical feature into a contiguous integer domain. Dummy-coding often follows binning, recoding, or feature hashing and transforms integers into sparse binary vectors with a 1 at the given position. These sparse output vectors require consolidating output column offsets. A recoding-build phase constructs a dictionary to map distinct values to integer codes (and optionally their frequency). The apply phase then uses the maps for transforming the given data. Feature hashing is less expensive (no dictionaries) and allows adjusting the output dimensions via domain size k. A data-parallel build phase requires managing partial hash-maps and merging.

Modality-specific Transformations: Besides the common transformations, modern ML systems provide high-level utilities for encoding data modalities like text and images. Two commonly used feature transformations for text are Bag of N-grams (Bag of Words) [81], and (word) embeddings [25, 68]. A Bag of N-grams representation tokenizes text, represents each n-gram (sequence of n tokens) by an integer ID, and counts the n-gram occurrences per sentence or document (potentially weighted by tf-idf). Another widely used transformations for text (and graphs) are embeddings, which map distinct tokens to numerical vectors. These embeddings are trained on data from the target domain. In contrast to text, images are stored as N-D tensors with multiple channels per pixel. Common image manipulations such as cropping, rotating, and adjusting brightness or contrast perform numerical transformations.

2.2 Implementations and Configurations

Existing ML systems provide built-in support for feature transformations, but with major differences in their user APIs, configurations, metadata management, and optimization techniques. In this section, we survey feature transformations in popular systems.

User APIs: In terms of the user APIs and metadata handling (e.g., dictionaries), we see two approaches. First, Scikit-learn [19, 94], Spark MLlib [99], and Keras pre-processing [51] expose separate APIs for build and apply (e.g. `fit` and `transform`). They embed the metadata inside the encoder objects and allow access via class attributes (e.g. `categories_` [93]). Users employ these objects to apply the encoders on unseen data. Second, TensorFlow (`feature_column` module [103]) and SystemDS [100] use a single API to transform an input dataset and manage the metadata in a stateless manner. For instance, SystemDS's `transformencode [100]` returns the metadata as a frame (allowing post-processing) and it provides a separate function, `transformapply [100]` to encode unseen data. Additionally, these frameworks provide methods to assemble multiple transformations for an entire dataset, either by pipelining (e.g. Scikit-learn, MLlib) or by a single API call with

a transform specification (e.g., SystemDS, TFX). Combining multiple transformations in a single pipeline creates optimization and parallelization opportunities, irrespective of the user API.

Configurations: All libraries provide certain tuning parameters to govern the encoded values and handle errors. First, common transformations might be fused into a single API call (e.g., Scikit-learn's *KBinsDiscretizer* [92] fuses binning and dummy-coding). Second, recoded values might be ordered alphabetical or by frequency. Third, users might provide fixed dictionaries (e.g., Keras `StringLookup` [104] `vocabulary` input). Fourth, common techniques to handle unknown categories include explicit exceptions, ignoring respective rows, and representing unknown or infrequent values as a NaN value or single overflow category.

Optimizations: Common parallelization strategies for feature transformations in ML systems include (1) parallel execution of feature encoders (column groups), (2) parallel execution on row partitions, and (3) pipelining of transformations and subsequent operator pipelines. First, Scikit-learn parallelizes feature transformations via Joblib [1] (Python's process-based parallelism) with one encoder per process. Second, Spark Mllib [67] and ML.NET [6] parallelize over row partitions or row cursors. Similarly, TensorFlow `tf.data` [38, 72] executes user-defined functions (UDFs) on data partitions in parallel, but without built-in support for feature transformations. Scikit-learn with Dask [23, 87] and Ray [71] backends as well as MLlib further execute distributed, data-parallel transformations on row partitions. Third, the TensorFlow dataset API enables overlapping data loading, transformations, and training via explicit mini-batch prefetching. Recently, NVIDIA RAPIDS [83] and DALI [40] introduced pre-processing on GPUs, which is challenging for irregular transformations of strings and sparse outputs. Overall, these strategies yield good runtime for simple transformations but are suboptimal for complex, multi-pass transformation workflows, and challenging data characteristics (many features/distinct items).

3 UPLIFT SYSTEM ARCHITECTURE

Our UPLIFT framework underpins specific transform built-in functions, which take a data frame and transform specification (JSON configuration) as input, and optimize and execute the transformations according to data characteristics. Inspired by future-based parallelization schemes [71] and query processing in column stores, UPLIFT creates and optimizes fine-grained task graphs as shown in Figure 1. In this section, we describe the task types, construction of task graphs, and their rule-based optimization in detail.

3.1 Task Types

We parse the transform specification, and create general and encoder-specific tasks. Unspecified features are handled by pass-through encoders, which casts those features to floating point.

Build: A *build* task scans an assigned feature of the input data frame and creates the necessary metadata. For recoding, this metadata is a dictionary of distinct items. In contrast, the binning metadata is an array of bin boundaries. For equi-width, we find the minimum and maximum values and arithmetically derive the bin boundaries. For equi-height, we sort the values and derive the bin boundaries from quantiles by position. Feature hashing and pass-through encoders do not collect metadata.

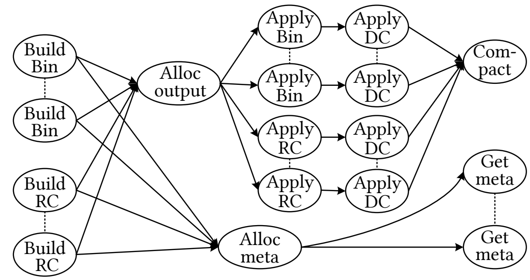


Figure 1: Task Graph for Adult Dataset.

Output Allocation: An *output allocation* task creates and allocates the output matrix. This upfront allocation allows the apply tasks to concurrently fill the output matrix. We determine the upper bound of the number of non-zero values and accordingly allocate a dense or sparse CSR (compressed sparse rows) matrix. For a CSR matrix, we already fill the row pointers and column indexes during allocation to avoid shifting and contention during concurrent apply.

Metadata Allocation: A *metadata allocation* task creates and allocates a frame for materializing all encoder's meta data. The dimensions of the metadata frame are the maximum metadata length (#distinct values, #bins) $\times$ #input-features. Pre-allocation allows again concurrent column-wise, metadata collection.

Apply: An *apply* task reads a feature from the input frame, encodes it using the metadata, and writes the encoded values into the output matrix. For cache-conscious operations from column-oriented frames to row-oriented matrices, we perform the apply task in a block-wise manner. According to output characteristics, we create dense or sparse apply tasks. Sparse tasks modify the indexes and values of a sparse, pre-allocated matrix in place. Recoding, binning and feature hashing insert into the first column of the output domain. A dummycode apply-task then converts the integer column of domain N into N binary codes. A pass-through apply-task parses inputs to floating-point values if necessary.

UDF Apply: For flexible transformation workflows, we further support UDF apply tasks. These UDFs are linear-algebra built-in or user functions called on columns. Internally, we extract the features, utilize `eval` function mechanisms to load, compile, and execute the UDFs, and write back the results. Examples include normalizing a subset of features, specific embedding strategies, and data cleaning primitives. This design enables reusing DSL-based built-in functions and system infrastructure for rewrites, code generation, and HW accelerators. However, the runtime compilation hinders estimating the output sparsity and thus, we allocate a dense output for UDFs.

Sparse Row Compaction: Missing values might be encoded as zeros, which leads to zeros in the sparse outputs after apply. A *compaction* task compacts sparse rows in-place by removing the zeros, shifting the non-zero entries, and updating offsets. We track the row indexes having zeros and compact only those rows.

Metadata Collection: A *metadata collection* task serializes the metadata—in an input-feature-aligned manner—into a frame.

3.2 Task-graph Construction

After splitting the transform specification into tasks, we consolidate all the tasks into a task-graph $\mathcal{G}$—which represents the parallelization strategy (execution plan). Later we submit this task-graph,

whose nodes are futures, to a `ForkJoinPool`, where tasks with ready inputs are amenable for execution (similar to TensorFlow's executor [3]). In addition to an array of tasks, we maintain an auxiliary map that incorporates the dependencies. The optimization objective of the task-graph construction is to minimize compute time ($C(\mathcal{G})$) under a given memory constraint ($\max M(\mathcal{G}) \leq M_B$).

$$\min\ C(\mathcal{G}) \quad \text{s.t. } \max(M(\mathcal{G})) \leq M_B \tag{1}$$

Figure 1 shows the initial task-graph for the transformation of a subset of features of the Adult dataset. We perform binning (Bin) for numerical and recoding (RC) for categorical columns, both followed by dummy-coding (DC). Arrows represent task dependencies and independent tasks can execute concurrently. We start with feature-wise build tasks, followed by output and metadata allocation tasks. Subsequently, we schedule the apply-tasks. Similarly, metadata collection tasks wait for metadata allocation (and thus, the build tasks). Finally, we have the compaction task of the sparse output.

3.3 Rule-based Optimizer

After constructing the global task-graph, we pass it through an optimizer. The optimizer first collects a uniform sample of rows to estimates the number of distinct items [41] and memory usage of the parallel tasks. Based on the memory estimates, data characteristics, and transform spec, the optimizer then rewrites the task-graph by updating the task array and the dependency map. Maintaining the dependencies in a single map simplifies new rewrites. Our current rewrites aim to reduce costs by eliminating synchronization bottlenecks, and better exploiting the available parallelism.

Reduce Bottlenecks: We scan the dependency map to remove unnecessary synchronization barriers. Examples include concurrent build, output allocation and metadata allocation tasks if the output dimensions are known prior to the build tasks (e.g. #bins).

Row Partitioning: Figure 1 shows column-wise build and apply tasks. Column-oriented task partitioning fails to fully utilize all cores if the input has fewer features than available cores, and if the compute time is skewed across columns (varying #distinct items, expensive pass-through casts). To exploit the available parallelism better, we additionally partition a column into multiple row-ranges and assign a task to each block of rows. For recoding, build tasks maintain private hash-maps, and a new merge task combines the partial maps via a union distinct (and aggregates). Similarly, for equi-width binning, every task finds the minimum and maximum values in the assigned partition. A merge task later derives the global aggregates from the partial values. For equi-height binning, each build task sorts a partition (run), which are later merged.

Number of Partitions: Depending on the number of distinct values and string sizes, increasing the number of row partitions (tasks operating on row ranges) increases memory overhead due private maps with overlap. We utilize a sample-based estimator [29, 41] for estimating the number of distinct items $\hat{d}$ ("generalized jackknife" estimator w.r.t. variability of frequencies), but scan-based estimators like KMV [13] and HyperLogLog [33, 44] could be used as well. From the average entry size in the sample and $\hat{d}$, we then derive the estimated size of partial hash-maps, allowing our optimizer to find a good number of partitions for each feature. Heuristically, we schedule more tasks than cores to mitigate skew. For the build and apply phases, we schedule 2 and $4 \times \#cores$ tasks, respectively. We also ensure a minimum number of rows in each partition to avoid unnecessary overhead, and we reduce the degree of parallelism if the total memory estimate exceeds the memory budget.

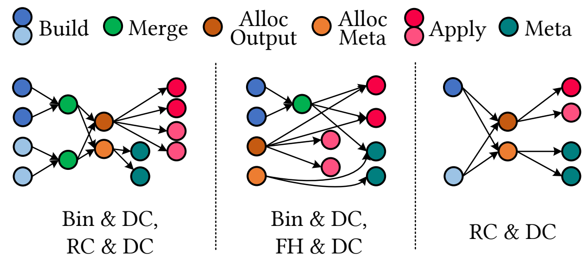


Figure 2: Three Examples of Optimized Task Graphs.

Example Plans: Figure 2 shows three examples (each with two features) of optimized parallelization strategies. In the first example (left), we apply binning and dummy-coding on a numerical column and recoding and dummy-coding on a categorical column. We schedule two partial build tasks per column, followed by their merge tasks. We then schedule output and metadata allocation tasks, the parallel apply tasks (two per column), and metadata collection tasks. In the second example (middle), we replace recoding with feature hashing. Here, the allocation tasks have no dependencies to the build tasks as the optimizer derives the output dimensions from #bins and the hash domain k. The third example (right) includes recoding and dummy-coding. Here, we schedule a single recode build-task per feature because otherwise, the estimated total size—including partial maps—exceeds the memory constraint (M_B).

3.4 Limitations

We summarize remaining limitations, which we see out-of-scope of the initial UPLIFT framework and thus, as future work.

- *Runtime Backends:* Our framework currently only applies to local operations on CPUs. Extending UPLIFT to distributed, data-parallel [118] operations, federated backends [10], and hardware accelerators [83] is interesting future work.
- *Optimizer Guarantees:* Our rule-based optimizer does not yet provide *guarantees* on finding cost-optimal plans, or ensuring not to exceed the given memory budget. Other valuable optimizer extensions include scan sharing among transformations, and fusing build and apply phases.

4 TRANSFORMATION BENCHMARK

As a basis for evaluating—and to foster research on—feature transformations, we define the FTBench benchmark using synthetic and publicly available real datasets. FTBench leverages well-known and publicly available data sets from UCI [27], AMiner [2] and Kaggle. Inspired by previously reported challenges [114], we augment the benchmark with real and synthetic datasets that capture choke points [30]. These datasets and use cases cover different domains and modalities (numerical, categorical, text, and time series), common feature transformations, varying data and transformation characteristics (number and distribution of distinct values, number of bins, string lengths, and sparsity), and workload types (batch and mini-batch transformation). We further define scale factors for selected use cases. Table 2 summarizes our 15 use cases. The

Table 2: Overview of FTBench Datasets and Use Cases.

ID	Dataset	Input Shape	Transformations	Significance	Output Shape
T1	Adult	32K × 15	Bin+DC (5), DC (9), PT (1)	Popular dataset	32K × 130
T2	KDD 98	95K × 469	Bin (334), DC (135), Scale (469)	Skewed #distinct: 50-900	95K × 6K
T3	Criteo	10M × 39	DC (26)	Skewed & large #distinct: 10-1.4M	10M × 5.8M
T4	Criteo	10M × 39	Bin (13), RC+Scale(26)	Scaled binning & #distinct	10M × 39
T5	Santander	200K × 200	Bin+DC (200)	Equi-height with small #bins	200K × 2K
T6	Crypto	48M × 10	Bin (10)	Large #bins (100K), equi-width	48M × 10
T7	Crypto	48M × 10	Bin (10)	Large #bins (100K), equi-height	48M × 10
T8	HomeCredit	31K × 122	DC (16)	Popular use case	31K × 245
T9	CatInDat	3M × 24	FH+DC (24)	Feature hashing for large #rows	3M × 24K
T10	Abstract	281K × 3	Count Vectorizer	Bag-of-Words w/ large #distinct	281K × 25M
T11	Abstract	100K × 1K	Embedding (dim = 300)	Embedding large #words	100K × 300K
T12	Synthetic	100K × 100	Bin (50), RC (50)	Mini-batch transformation	100K × 100
T13	Synthetic	10M × 10	RC (10)	Varying *strlen*: 25-500	10M × 10
T14	Synthetic	100M × 4	RC (4)	Varying #distinct: 100K-1M	100M × 4
T15	Criteo	5M × 39	Various Combinations	End-to-end feature engineering	Scalar

transformations column indicates the applied transformation types and the number of columns they are applied on.

Adult is a commonly-used census dataset for classifying high-/low salary with 6 numerical and 9 categorical features. Use case T1 dummy-codes 9 categorical columns, as well as bins (5 equi-width bins) and dummy-codes 5 numerical columns.

KDD 98 is a regression problem for donation campaign returns. This dataset has 334 numerical and 135 categorical features with skewed distinct values across columns (in the range of 50 to 900). In this T2 use case, we first bin (5 equi-width bins) the numerical columns, and dummy-code both the binned and categorical columns, followed by standard scaling on all columns.

CriteoD21: We use 1 of 24 days from the Criteo [56] click logs dataset, which consists of 13 numerical and 26 categorical features. The categorical columns have skewed distinct values (10 to 76M) and string entries are 8-character hashes. We define two use cases: T3 performs dummy-coding on the categorical columns, and T4 performs binning on numerical features (equi-width 10 bins) and recoding and standard-scaler on categorical features. Additionally, we define a scale factor to select a subset of the rows from 1M to 192M with a default value of 10. With 10M rows, the first use case produces 5.8M features after dummy-coding. These use cases study dummy-coding and binning with many rows and columns.

Santander [89] from Kaggle contains an anonymized transaction history of the Santander bank for predicting transactions. It has 200 numerical columns and 200K rows. Use case T5 applies equi-height binning (10 bins) and dummy-coding on all the features.

Crypto [34] from Kaggle contains historic trades of crypto assets at a minute granularity. This dataset has 24M rows and 10 numerical features. We define two binning use cases (T6 with equi-width, T7 with equi-height) both with 100K bins for every column. Additionally, we define a scale factor to increase the number of rows (default 2 produces 48M rows) via replication.

Home Credit: The Home credit default risk dataset [39] contains information on loan re-payments. This dataset has 31K rows and 122 columns, out of which 16 are categorical. Use case T8 dummy-codes the categorical columns into 245 output columns.

Cat in Dat: The Categorical Feature Encoding Challenge [48] from Kaggle tests various categorical feature encoding techniques. This dataset has 24 categorical features and 300K rows with #distinct between 2 and 300K. We replicate the dataset by a scale factor (default 10 produces 3M rows). Full dummy-coding produces 316K columns. However, in use case T9, we apply feature hashing (with $k = 1\text{K}$) followed by dummy-coding, which results in 24K features.

Paper Abstracts: Furthermore, we use the paper abstracts from the AMiner citation dataset [2]. Use case T10 tokenizes the abstracts into unigrams, bigrams, and trigrams, and creates Bags of N-grams. Tokenizing 281K abstracts produces 93M N-grams, out of which 25.5M (473K #unigrams, 6M #bigrams, 19M #trigrams) are unique. Use case T11 simulates scoring with pre-trained word embeddings. We perform word embedding (pre-trained on Wikipedia, dimension 300) on the abstracts—padded to the maximum abstract length—in a batch-wise manner (10K batch size). Embedding 100K abstracts padded to maximum length of 1K words yields a 100K×300K matrix, which has several challenging choke points as well [42].

Mini-Batch Transforms: T12 defines mini-batch feature transformations, typical for Deep Neural Networks (DNNs) and scoring. We use a synthetic dataset of shape 100K × 100 (50 numerical and 50 categorical). To simulate mini-batch scenarios, we execute 10 epochs with batch size 1024, where we apply encoders for equi-width binning and recoding, as well as $\max(\mathbf{MV})$ operations.

String Length: T13 tests the impact of string lengths on recoding. The synthetic data has 10M rows and 10 categorical columns, each with 1M distinct values. Entries are generated as fixed-length, random alphanumeric strings. A large number of distinct values makes it more difficult to keep the hash-maps in the CPU caches. A scale factor varies the string length in $[25, 500]$ (default 200).

Distinct Values: T14 studies the impact of a large number of distinct values. We apply recoding on a synthetic dataset having 100M rows and 4 columns. All feature values are 5-character alphanumeric strings. A scale factor controls the number of distinct values from 100K to 10M (default 1M) per feature.

Feature Engineering: Finally, T15 represents a feature engineering use case [45, 50]. We encode the Criteo dataset (scale factor 5) with different feature transformation specifications, and find the best configuration using Naive Bayes as an inexpensive estimator. Used transformations include binning with different numbers of bins, and different categorical encoders.

5 EXPERIMENTS

We study the performance of UPLIFT for various feature transformations and data characteristics via micro benchmarks and our FTBench feature transformation benchmark. Overall, we observe that optimized parallelization strategies yield robust improvements.

5.1 Experimental Setting

HW Environment: We ran all experiments on a Ubuntu 20.04.1 node having a single AMD EPYC 7302 CPU @3.0-3.3 GHz (16 physical/32 virtual cores) with 512KB, 8MB and 128MB L1, L2 and L3 caches, and 128 GB DDR4 RAM (peak performance is 768 GFLOP/s, 183.2 GB/s). We used OpenJDK 11 with 110 GB max and initial JVM heap sizes for UPLIFT and Python 3.8 for other baselines.

Baselines: We compare UPLIFT with multiple popular ML systems and frameworks under different configurations.

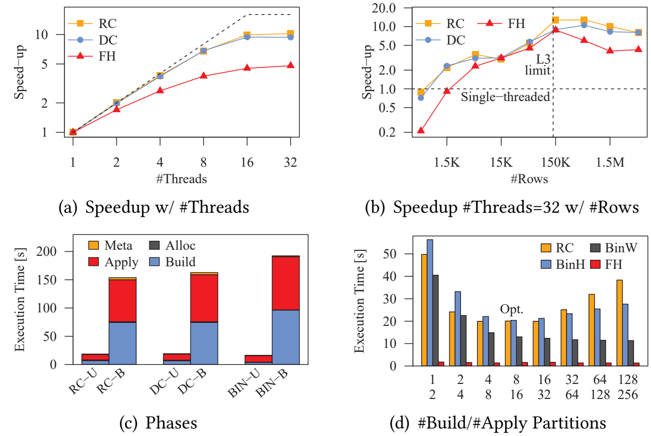
- *Apache SystemDS [15]:* Our primary baseline is **Base**, which refers to the default configuration of SystemDS with single-threaded *transformencode*. Base, however, shares the new cache-conscious runtime implementation with UPLIFT.
- *Scikit-learn [76]:* **SKlearn** is the most popular choice for feature transformations. We use its *FeatureUnion* to construct pipelines for transformations available in the *Preprocessing* module. We also explored its *n_jobs* parameter for parallelism, but did not measure significant improvements.
- *Other ML Systems:* Additionally, we use more specialized ML Systems for some use cases. These systems include Dask [87] and spark.ml 3.1 [67] (**Spark**) for row-based parallelization and fused transformation/training pipelines, as well as Keras [21] on Tensorflow 2.8 [3] for NLP use cases.

We provide two reference implementations of the full proposed benchmark (see reproducibility): one for UPLIFT and Base—written in SystemDS' DML scripting language with R-like syntax [15] and JSON transform spec—and another for SKlearn. We warm start SystemDS with two runs for JIT compilation. We report the average elapsed time of the next 3 runs for SystemDS and Sklearn.

5.2 Micro Benchmarks

We conduct an ablation study of various aspects of UPLIFT. The micro benchmarks investigate speedup, time breakdown of phases, impact of different numbers of row partitions, and the UDF-based integration of linear algebra operations. We use synthetic datasets for controlling data characteristics, distinct values, and string lengths.

Speedup: We first evaluate the speedup of UPLIFT with increasing #threads and #rows for recoding (RC), dummy-coding (DC), and feature hashing (FH, with k = 10K). We use synthetic data with 5M rows, 100 categorical columns (with 100K #distinct each), and 5-char strings, where dictionaries still fit in L3. Figure 3(a) shows the speedup with increasing #threads of up to 10x at 16 physical cores. Despite producing an ultra-sparse output with 10M columns, DC performs equally well due to efficient sparsity handling. In contrast, FH shows a smaller speedup because it is a memory-bandwidth-bound operation. Figure 3(b) further shows the speedup—compared to single-threaded operations—with increasing number of rows, while keeping the #distinct constant at 10K. UPLIFT yields improvements from 1.5K rows and the highest speedup for RC (12x) and DC (10x) at 500K rows, and for FH (8x) at 150K rows.



(a) Speedup w/ #Threads (b) Speedup #Threads=32 w/ #Rows

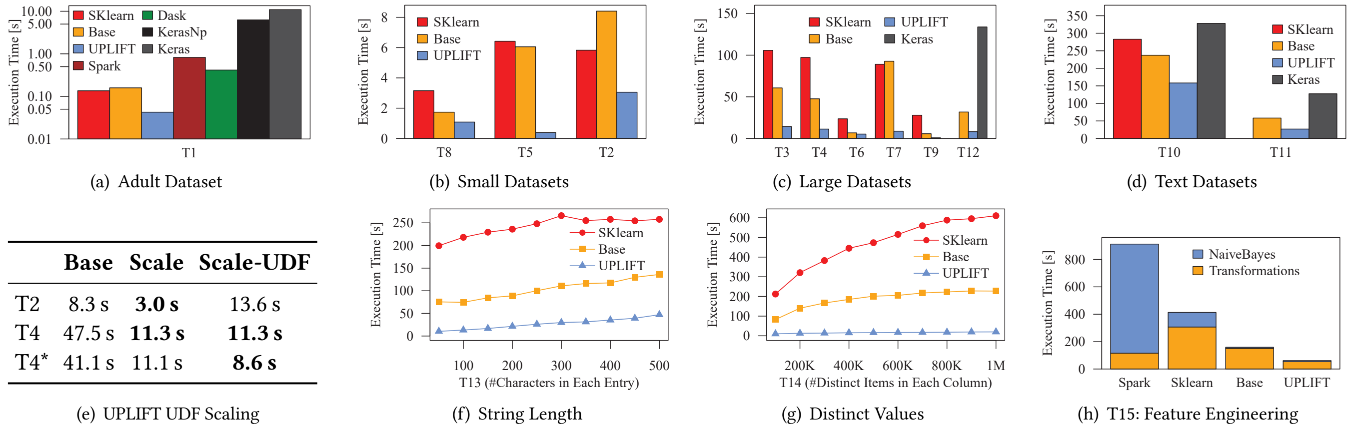
(c) Phases (d) #Build/#Apply Partitions

Figure 3: Micro Benchmarks.

Transformation Phases: To understand the absolute time and speedup of the individual transformation tasks, we measure their time break-down. We use the same dataset as before but place a barrier after each phase in the task-graph. Figure 3(c) shows the breakdown for UPLIFT (XX-U) and Base (XX-B). For binning, we replace 50 categorical by numerical features. For RC and DC, the apply phase is slightly slower than the build. This is because apply requires the threads to read from a column-oriented data frame and write to a row-oriented matrix, which causes cache misses. All input features were strings, whose parsing overhead for binning can be reduced via schema information on read. The sparse DC-apply (into a CSR matrix) is slightly slower than the dense RC-apply because column-oriented access into the row-pointer arrays of a CSR matrix causes more cache misses. Finally, the overheads of allocations and multi-threaded metadata collection are negligible here.

Row Partitioning: We further study the impact of the number of partitions (tasks) for build and apply on synthetic data of 100M rows, 4 columns of 1M distinct values (5 chars) each. In detail, we evaluate 8 configurations of build and apply for recoding (RC), feature hashing (FH), equi-width (BinW, 10), and equi-height (BinH, 10) binning. Starting with 1/2 partitions for build and apply, we double them step-wise up to 128/256. Figure 3(d) shows that performance improves up to 8/16. Beyond that, the overhead of partial maps and their merging increases. Feature hashing is robust to task over-provisioning because it has no metadata. Our heuristic-based optimizer also picks 8 and 16 #partitions for this dataset.

Linear Algebra Operations: There are two approaches of integrating linear-algebra-based transformations with UPLIFT: UDF apply tasks and LA programs outside UPLIFT (our default). Figure 4(e) shows the execution time for both and Base on FTBench's T2 and T4 as well as a modified T4 (T4*), where we replace Binning with min/max-scaling. For T2, UPLIFT with std-scaling as UDF is 4.5x slower than calling std-scaling as a built-in after `transformencode`. Forcing a dense output matrix and column-oriented scaling of a row-oriented matrix reduces performance. In contrast, a separate scaling exploits SystemDS' row-wise multi-threaded operations. For T4, however, both approaches perform equally well, as the output matrix by recoding is already dense. For T4*, the UDF approach yields a 30% improvement because it reduces synchronization barriers and produces fewer materialized intermediates.



	Base	Scale	Scale-UDF
T2	8.3 s	**3.0 s**	13.6 s
T4	47.5 s	**11.3 s**	**11.3 s**
T4*	41.1 s	11.1 s	**8.6 s**

(e) UPLIFT UDF Scaling

Figure 4: FTBENCH Use Case Results and Ablation Studies.

5.3 Feature Transformation Benchmark

We now report FTBENCH benchmark results, primarily for SystemDS (Base and UPLIFT) and SKlearn. The SKlearn pipelines (`FeatureUnion`) have been created automatically by parsing the same JSON transform spec used for UPLIFT and Base. We divide the use cases into five groups: small real datasets, large real datasets, text transformations on real datasets, large strings and many distinct values on synthetic data, and other baselines. Finally, we execute the end-to-end feature engineering use case and a subset of other use cases in two specialized systems and report the results.

Small Datasets: Figure 4(a) shows the runtime of all baseline systems for the use case T1. For the Dask, Spark and Keras implementations, we build a pipeline of transformations with lazy evaluation. We compare two Keras methods for dictionary creation: Keras' built-in `adapt` (Keras) and `Numpy.unique()` to find the distinct items and pass it to Keras (KerasNp). KerasNp is 2x faster than Keras; Base and SKlearn are similar and 32x/52x faster than KerasNp and Keras. UPLIFT further improves the runtime by 6x. Dask and Spark's static parallelization schemes are ineffective for smaller datasets, and ≈10x slower than UPLIFT. Figure 4(b) shows the runtime for the remaining use cases on smaller datasets. For T8, UPLIFT improves by 25% and 2.3x over Base and SKlearn. Pass-through (106 columns) with provided schema is memory-bandwidth-bound and only benefits slightly from multi-threading. For T5, UPLIFT is 15x and 16x faster than Base and SKlearn. The build phase of equi-height binning benefits from parallelization (parallel sort). For T2, SKlearn is 25% faster than Base. UPLIFT further improves by 2x and 2.6x. For this use case, we keep the row-wise, multi-threaded standard scaling outside of `transformencode`/UPLIFT.

Large Datasets: Figure 4(c) shows the runtime for the large-scale use cases. T3 and T4 are based on the Criteo dataset (scale factor 10), where UPLIFT yields speedups of 4x/7x for T3 and 4x/9x for T4 compared to Base and SKlearn. T6 and T7 use the Crypto dataset (scale factor 2). For equi-width binning in T6, UPLIFT is 2x and 6.5x faster than Base and SKlearn, where both Base and UPLIFT find the bin boundaries via a single scan. UPLIFT improves by 12.4x and 11.8x for T7 by parallelizing the more expensive build phase of equi-height binning. For T9 (scale factor 10), UPLIFT yields speedups of 7x and 31x over Base and SKlearn. In T12, we replace SKlearn with Keras. UPLIFT is 3.8x and 16x faster than Base and Keras. While the Keras apply phase is 2x faster than Base, its build phase is 10x slower. In these larger use cases, Base generally performs better due to the cache-conscious runtime of `transformencode`.

Table 3: Comparison with Other Baseline ML Systems.

	Spark	Spark1T	Dask	SKlearn	Base	UPLIFT
T2	19.6 s	48.4 s	99 s	5.8 s	8.3 s	**3 s**
T3	44.2 s	133.4 s	80.6 s	105.7 s	62 s	**14 s**
T9	**0.75 s**	1.3 s	NA	27.9 s	6.1 s	**0.85 s**

Text Transformations: Figure 4(d) shows the runtime of T10 and T11 (text). We use the *nltk* [102] Python library for tokenizing n-grams. For T10, we read the n-grams, recode the token sequence, and construct a selection matrix via `table` from sequence positions to distinct tokens. UPLIFT is 33% faster than Base due to the parallelized recode, and 1.7x/2x faster than Sklearn's `CountVectorizer` and Keras' `TextVectorization`. For T11, we use `transformapply` to convert the tokens into a selection matrix and obtain the embeddings via a matrix multiply. UPLIFT uses multi-threaded operations and is 2x and 4.5x faster than Base's single-threaded operations, and Keras-Tensorflow's embedding layer. This result shows that transformations via LA operations yield very competitive performance by reusing runtime kernels and sparsity handling [98].

Data Characteristics: Furthermore, we systematically vary the string lengths in T13, and the #distinct per column in T14. Figure 4(f) shows the T13 results, where UPLIFT is 7.5x faster than Base for strings of size 50. However, the speedup drops to 2.9x for string length 500 due to more cache misses. Similarly, UPLIFT improves over SKlearn by 21x for smaller and 5x for the larger strings. In contrast, Figure 4(g) shows increasing speedup with an increasing #distinct (row-partitioned parallel recode-build). UPLIFT improves over Base and SKlearn by 9x/20x for 100K and by 11.4x/30x for 1M distinct values due to multi-threaded latency hiding.

Other Baselines: Table 3 shows the execution time of T2, T3 and T9 for more baselines: Spark, single-threaded Spark (Spark1T), Dask, and SKlearn. We replaced dummy-coding with recoding in Dask due to its sub-par sparsity handling. Spark and Dask yield 2.4x and 24% speedups compared to SKlearn for T3. However, UPLIFT's dynamic parallelization scheme is 3x and 5.6x faster than Spark and Dask. For the small T2, Spark improves 2.5x over single-threaded execution,

but is 6.7x slower than UPLIFT. For T9, Spark and UPLIFT performs similarly. Spark compresses the sparse features into a single column for feature hashing, which avoids the handling of sparsity. Dask does not provide a feature-hashing API. Figure 4(h) compares the runtime of feature engineering in T15, with a time breakdown of 6 different feature transformations and their downstream Naïve Bayes training. UPLIFT yields speedups of 2x/3x/6x over Spark, Base and SKlearn for transformations; and 13x/2.3x/5.3x overall.

6 RELATED WORK

Our UPLIFT framework and benchmark for parallel feature transformation workloads are generally related to feature engineering and ML system benchmarks, but also to specific techniques for task scheduling and efficient group-by/join operators.

Feature Engineering: As part of data preparation, data scientists engineer general or domain-specific features [4, 88], perform feature transformations, and apply feature selection [107, 111, 120]. Besides our survey of feature transformations in Section 2, there are related system categories. First, feature-centric tools like DeepDive [96], Overton [86], and Ludwig [70] focus on multi-modal input features, and provide, for example, high-level, data-type-specific encoders, combiners, and decoders [70]. Second, feature engineering is also related to data validation [80, 90] (e.g., with checks of the # distinct items for categoricals and min/max ranges for numerics) as well as data augmentation [84, 110] (where labeling functions are defined on high-level features). An organization-wide reuse of data validation constraints and features (e.g., through feature stores [75]) further ensures consistency and avoid unnecessary redundancy. Third, several AutoML tools such as Auto-WEKA [55, 106] and Auto-sklearn [31, 32] include data preparation, feature transformations, and/or feature selection primitives in order to improve end-to-end accuracy. Similarly, sparse n-gram token featurization [42] finds relevant n-grams for pruning, and FairExp [88] constructs and selects features for bias reduction; both, without significant accuracy degradation. In contrast to these feature-oriented systems—which simplify finding good transformations—our UPLIFT framework makes an orthogonal contribution of improving the runtime of given transformations via fine-grained task and data parallelism.

Benchmarking ML Systems: Early benchmarks for ML systems were part of large-scale data analysis benchmarks such as BigBench [20, 36], HiBench [115], GenBase [101], and SparkBench [5], often defining ML pipelines including the necessary feature transformations. The new TPCx-AI [109] follows similar design principles with scaling up to 10TB. Later DNN benchmarks such as DAWNBench [22] and MLPerf [66] focused specifically on training computer vision, translation, and recommendation models to a specific target accuracy. More recently, a variety of benchmarks have been proposed for special aspects such as AutoML [37, 63], linear algebra operations [105], data cleaning [60], classifying feature types [95], and multi-task reinforcement learning [117]. Our FTBench fills the gap of systematically evaluating feature transformations.

Task-based Scheduling: Parallel feature transformations mostly rely on static, parallelization strategies [1, 6, 67], overlay transformations and model training [4, 72] and delayed materialization [6]. However, fine-grained task scheduling has been studied extensively in different areas. First, task-parallel and hybrid (task- and data-parallel) strategies in ML systems includes future-based task scheduling in Ray [71], parallel for-loops in SystemDS [17], inter- and intra-operator parallelism in TensorFlow [3], irregular data partitioning/decomposition in DMac [116], Dask [87], and Modin [77, 78], flattened nested parallelism in Matryoshka [35], and model-hopper parallelism in Cerebro [73]. Several systems also leverage task partitioning and scheduling from HPC [46, 54]. Second, fine-grained task parallelism has also been leveraged in query optimization [43, 97]. Similar to fine-grained transformation tasks, breaking operators into basic primitives can create more optimization potential [26, 28, 74]. Fourth, at runtime level, fine-grained task scheduling gains increasing popularity. Examples are NUMA-aware operator and pipeline scheduling [52, 59, 108], prioritization of HTAP workload tasks [82, 112], and irregular reinforcement learning workloads with Ray [61, 71]. UPLIFT similarly schedules fine-grained task graphs, but optimizes these graphs according to data, (transformation) workload, and cluster characteristics.

Group-by and Join Operators: Feature transformation tasks—like recoding build and apply—are closely related to efficient group-by (or deduplication) and join operators. There is a long history of comparing hash and sort-merge join implementations on contemporary hardware [7, 8, 53, 91]. In micro-benchmarks, partitioned hash-joins [65]—with one or more range/radix partitioning phases for better locality—often perform very well. However, comparisons on query workloads observed slow-downs with partitioning due to pipeline breakers and out-of-order tuple fetching [9]. Recently, Bloom and Cuckoo filters are increasingly used—through sideways information passing [47]—for pre-filtering of the probe side [9, 57]. Both, group-by and join operators have been specialized for GPUs [49], and such implementations were evaluated with new link technologies [64, 85]. Similarly, there are specialized implementations for compressed data [58] and group-joins with shared group-by and join keys [69]. In contrast to efficient group-by and join operators (except special handling of GROUPING SETS), feature transformations apply many concurrent build (group-by, sort) and apply (FK-PK join) operations. However, underlying ideas of group-by and join operators can be adapted to the specific characteristics of feature transformations (e.g., partitioned recode-build).

7 CONCLUSIONS

To summarize, we introduced UPLIFT as a parallel feature transformation framework with fine-grained task scheduling, as well as FTBench for evaluating such frameworks on a variety of transformation use cases. Breaking the transformation workload into fine-grained tasks and their dependencies yields a very clean framework that is amenable to optimization with regard to data, workload, and hardware characteristics. Our heuristic, transformation-based optimizer already achieves good improvements compared to static parallelization and baseline systems. During the development of UPLIFT, FTBench already proved to be very useful, and we hope it will foster additional research on efficient feature transformations. Interesting directions for future work include improved runtime techniques, a cost-based optimizer for feature transformation task graphs, and the pushdown of compression into such feature transformation pipelines. Finally, FTBench can be improved by additional reference implementations for more baseline ML systems.

REFERENCES

[1] 2021. Joblib: running Python functions as pipeline jobs. https://joblib.readthedocs.io/en/latest/ Accessed: 2022-07-11.
[2] 2022. Citation Network Dataset. https://www.aminer.org/citation Accessed: 2022-07-11.
[3] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
[4] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. 2017. Mind the Gap: Bridging Multi-Domain Query Workloads with Empty-Headed. *PVLDB* 10, 12 (2017), 1849–1852.
[5] Dakshi Agrawal et al. 2015. SparkBench - A Spark Performance Testing Suite. In *TPCTC@VLDB Workshop*.
[6] Zeeshan Ahmed et al. 2019. Machine Learning at Microsoft with ML.NET. In *SIGKDD*. 2448–2458.
[7] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB* 5, 10 (2012), 1064–1075.
[8] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB* 7, 1 (2013), 85–96.
[9] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD*. 168–180.
[10] Sebastian Baunsgaard et al. 2021. ExDRa: Exploratory Data Science on Federated Raw Data. In *SIGMOD*. 2450–2463.
[11] Denis Baylor et al. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *SIGKDD*. 1387–1395.
[12] Denis Baylor et al. 2019. Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform. In *OpML*. 51–53.
[13] Kevin S. Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. 2007. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*. 199–210.
[14] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.
[15] Matthias Boehm et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*.
[16] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768.
[17] Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, Prithviraj Sen, Yuanyuan Tian, Douglas Burdick, and Shivakumar Vaithyanathan. 2014. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB* 7, 7 (2014), 553–564.
[18] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. 225–237.
[19] Lars Buitinck et al. 2013. API design for machine learning software: experiences from the scikit-learn project. In *LML@PKDD Workshop*.
[20] Paul Cao, Bhaskar Gowda, Seetha Lakshmi, Chinmayi Narasimhadevara, Patrick Nguyen, John Poelman, Meikel Poess, and Tilmann Rabl. 2016. From BigBench to TPCx-BB: Standardization of a Big Data Benchmark. In *TPCTC@VLDB Workshop*.
[21] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras Accessed: 2022-07-11.
[22] Cody Coleman et al. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Oper. Syst. Rev.* 53, 1 (2019), 14–25.
[23] Dask Development Team. 2016. *Dask: Library for dynamic task scheduling*. https://dask.org Accessed: 2022-07-11.
[24] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*. 1701–1716.
[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. 4171–4186.
[26] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR*.
[27] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml Accessed: 2022-07-11.
[28] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*.
[29] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2018. Compressed linear algebra for large-scale machine learning. *VLDB J.* 27, 5 (2018), 719–744.
[30] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.
[31] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-Sklearn 2.0: The Next Generation. *CoRR* abs/2007.04074 (2020).
[32] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: Efficient and Robust Automated Machine Learning. In *Automated Machine Learning*. 113–134.
[33] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*.
[34] G-Research. 2021. G-Research Crypto Forecasting. https://www.kaggle.com/c/g-research-crypto-forecasting/data Accessed: 2022-07-11.
[35] Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. The Power of Nested Parallelism in Big Data Processing. In *SIGMOD*. 605–618.
[36] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD*. 1197–1208.
[37] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. 2019. An Open Source AutoML Benchmark. *CoRR* abs/1907.00909 (2019).
[38] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. 2022. Cachew: Machine Learning Input Data Processing as a Service. In *USENIX ATC*. 689–706.
[39] Home Credit Group. 2018. Home Credit Default Risk. https://www.kaggle.com/c/home-credit-default-risk/data Accessed: 2022-07-11.
[40] Joaquin Anton Guirao et al. 2019. Fast AI Data Preprocessing with NVIDIA DALI. https://developer.nvidia.com/blog/fast-ai-data-preprocessing-with-nvidia-dali Accessed: 2022-07-11.
[41] Peter J. Haas and Lynne Stokes. 1998. Estimating the Number of Classes in a Finite Population. *J. Amer. Statist. Assoc.* 93 (1998), 1475–1487.
[42] John Hallman. 2021. Efficient Featurization of Common N-grams via Dynamic Programming. https://sisudata.com/blog/efficient-featurization-common-n-grams-via-dynamic-programming Accessed: 2022-07-11.
[43] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. 2008. Parallelizing query optimization. *PVLDB* 1, 1 (2008), 188–200.
[44] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*. 683–692.
[45] Franziska Horn, Robert Pack, and Michael Rieger. 2019. The autofeat Python Library for Automatic Feature Engineering and Selection. *CoRR* abs/1901.07329 (2019).
[46] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. 1991. Factoring: a practical and robust method for scheduling parallel loops. In *Supercomputing*. 610–632.
[47] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *ICDE*. 774–783.
[48] Kaggle. 2019. Categorical Feature Encoding Challenge. https://www.kaggle.com/c/cat-in-the-dat/data Accessed: 2022-07-11.
[49] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS@VLDB Workshop*.
[50] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. 2016. ExploreKit: Automatic Feature Generation and Selection. In *ICDM*. 979–984.
[51] Keras development team. 2022. Working with preprocessing layers. https://keras.io/guides/preprocessing_layers/ Accessed: 2022-07-11.
[52] David Kernert, Wolfgang Lehner, and Frank Köhler. 2016. Topology-aware optimization of big sparse matrices and matrix multiplications on main-memory systems. In *ICDE*. 823–834.
[53] Changkyu Kim et al. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB* 2, 2 (2009), 1378–1389.
[54] Jonas H. Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M. Ciorba. 2022. LB4OMP: A Dynamic Load Balancing Library for Multithreaded Applications. *IEEE Trans. Parallel Distributed Syst.* 33, 4 (2022), 830–841.
[55] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA. *J. Mach. Learn. Res.* 18 (2017), 25:1–25:5.
[56] Criteo AI Lab. 2020. Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/ Accessed: 2022-07-11.
[57] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. 2019. Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput. *PVLDB* 12, 5 (2019), 502–515.
[58] Jae-Gil Lee et al. 2014. Joins on Encoded and Partitioned Data. *PVLDB* 7, 13 (2014), 1355–1366.
[59] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. 743–754.
[60] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE*. 13–24.

[61] Eric Liang et al. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *ICML*.
[62] Edo Liberty et al. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. 731–737.
[63] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. 2018. ML-Bench: Benchmarking Machine Learning Services Against Human Experts. *PVLDB* 11, 10 (2018), 1220–1232.
[64] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *SIGMOD*. 1633–1649.
[65] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730.
[66] Peter Mattson et al. 2020. MLPerf Training Benchmark. In *MLSys*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.).
[67] Xiangrui Meng et al. 2016. MLlib: Machine Learning in Apache Spark. *JMLR* 17 (2016), 34:1–34:7.
[68] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR Workshop*.
[69] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB* 4, 11 (2011), 843–851.
[70] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. *CoRR* abs/1909.07930 (2019).
[71] Philipp Moritz et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577.
[72] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *PVLDB* 14, 12 (2021), 2945–2958.
[73] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *PVLDB* 13, 11 (2020), 2159–2173.
[74] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.
[75] Laurel J. Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. 2021. Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems. *PVLDB* 14, 12 (2021), 3178–3181.
[76] Fabian Pedregosa et al. 2011. Scikit-learn: Machine Learning in Python. *JMLR* 12 (2011), 2825–2830.
[77] Devin Petersohn et al. 2020. Towards Scalable Dataframe Systems. *PVLDB* 13, 11 (2020), 2033–2046.
[78] Devin Petersohn, Dixin Tang, Rehan Sohail Durrani, Areg Melik-Adamyan, Joseph Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *PVLDB* 15, 3 (2021), 739–751.
[79] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. 1426–1439.
[80] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data Management Challenges in Production Machine Learning. In *SIGMOD*. 1723–1726.
[81] Fotis Psallidas et al. 2019. Data Science through the looking glass and what we found there. *CoRR* abs/1912.09536 (2019).
[82] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *PVLDB* 10, 2 (2016), 37–48.
[83] Sebastian Raschka, Joshua Patterson, and Corey Nolet. 2020. Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Inf.* 11, 4 (2020), 193.
[84] Alexander Ratner, Stephen H. Bach, Henry R. Ehrenberg, Jason Alan Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *PVLDB* 11, 3 (2017), 269–282.
[85] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR*.
[86] Christopher Ré. 2020. Overton: A Data System for Monitoring and Improving Machine-Learned Products. In *CIDR*.
[87] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *SciPy*. 130 – 136.
[88] Ricardo Salazar, Felix Neutatz, and Ziawasch Abedjan. 2021. Automated Feature Engineering for Algorithmic Fairness. *PVLDB* 14, 9 (2021), 1694–1702.
[89] Banco Santander. 2019. Santander Customer Transaction Prediction. https://www.kaggle.com/c/santander-customer-transaction-prediction/data Accessed: 2022-07-11.
[90] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *PVLDB* 11, 12 (2018), 1781–1794.
[91] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*. 1961–1976.
[92] Scikit-learn development team. 2022. sklearn.preprocessing.KBinsDiscretizer. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.KBinsDiscretizer.html Accessed: 2022-07-11.
[93] Scikit-learn development team. 2022. sklearn.preprocessing.OrdinalEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OrdinalEncoder.html#sklearn.preprocessing.OrdinalEncoder Accessed: 2022-07-11.
[94] Scikit-learn development team. 2022. User Guide for sklearn.preprocessing Package. https://scikit-learn.org/stable/modules/preprocessing.html Accessed: 2022-07-11.
[95] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. In *SIGMOD*. 1584–1596.
[96] Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. 2015. Incremental Knowledge Base Construction Using DeepDive. *PVLDB* 8, 11 (2015), 1310–1321.
[97] Mohamed A. Soliman et al. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD*. 337–348.
[98] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *SIGMOD*. 1607–1623.
[99] spark.ml development team. 2022. Extracting, transformation and selecting features. https://spark.apache.org/docs/latest/ml-features Accessed: 2022-07-11.
[100] SystemDS development team. 2022. DML Language Reference. https://apache.github.io/systemds/site/dml-language-reference Accessed: 2022-07-11.
[101] Rebecca Taft, Manasi Vartak, Nadathur Rajagopalan Satish, Narayanan Sundaram, Samuel Madden, and Michael Stonebraker. 2014. GenBase: a complex analytics genomics benchmark. In *SIGMOD*. 177–188.
[102] NLTK Team. 2022. Natural Language Toolkit. https://www.nltk.org/index.html Accessed: 2022-07-11.
[103] Tensorflow development team. 2022. Tensorflow feature column Module. https://www.tensorflow.org/api_docs/python/tf/feature_column Accessed: 2022-07-11.
[104] Tensorflow development team. 2022. tf.keras.layers.StringLookup. https://www.tensorflow.org/api_docs/python/tf/keras/layers/StringLookup Accessed: 2022-07-11.
[105] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-based Analytics. *PVLDB* 11, 13 (2018), 2168–2182.
[106] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms. In *SIGKDD*. 847–855.
[107] Robert Tibshirani. 1996. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society* 58, 1 (1996), 267–288.
[108] Pinar Tözün and Helena Kotthaus. 2019. Scheduling Data-Intensive Tasks on Heterogeneous Many Cores. *IEEE Data Eng. Bull.* 42, 1 (2019), 61–72.
[109] TPC. 2022. *TPCx-AI Specification, Version 1.0.2*. https://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-ai_v1.0.2.pdf Accessed: 2022-07-11.
[110] Paroma Varma and Christopher Ré. 2018. Snuba: Automating Weak Supervision to Label Training Data. *PVLDB* 12, 3 (2018), 223–236.
[111] William N. Venables and Brian D. Ripley. 2002. *Modern applied statistics with S, 4th Ed.* Springer.
[112] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. 2015. Extending database task schedulers for multi-threaded application code. In *SSDBM*. 25:1–25:12.
[113] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460.
[114] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD*. 2639–2652.
[115] Lan Yi and Jinquan Dai. 2013. Experience from Hadoop Benchmarking with HiBench: From Micro-Benchmarks Toward End-to-End Pipelines. In *WBDB*. 43–48.
[116] Lele Yu, Yingxia Shao, and Bin Cui. 2015. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*. 93–105.
[117] Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. 2019. Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning. In *CoRL*. 1094–1100.
[118] Matei Zaharia et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.
[119] Matei Zaharia et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.
[120] Ce Zhang, Arun Kumar, and Christopher Ré. 2014. Materialization optimizations for feature selection workloads. In *SIGMOD*. 265–276.