University of Bamberg

UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads

Arnab Phani, Lukas Erlbacher, Matthias Boehm

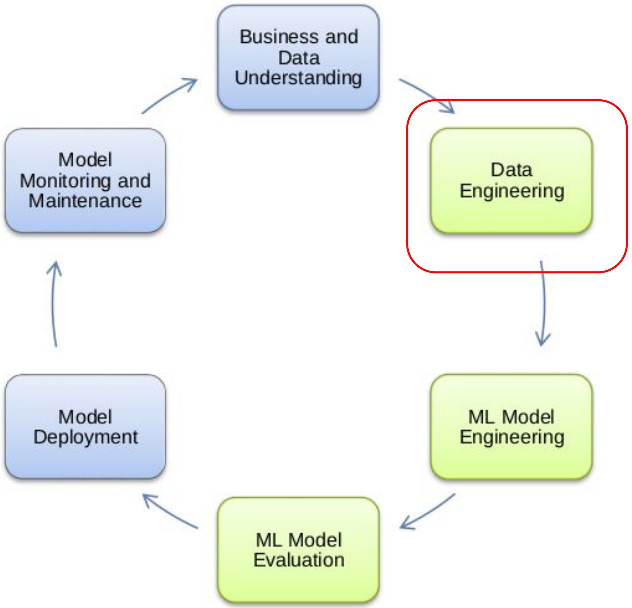
Liia Sharipova
Presenter for DT-DB42-M Seminar

Agenda

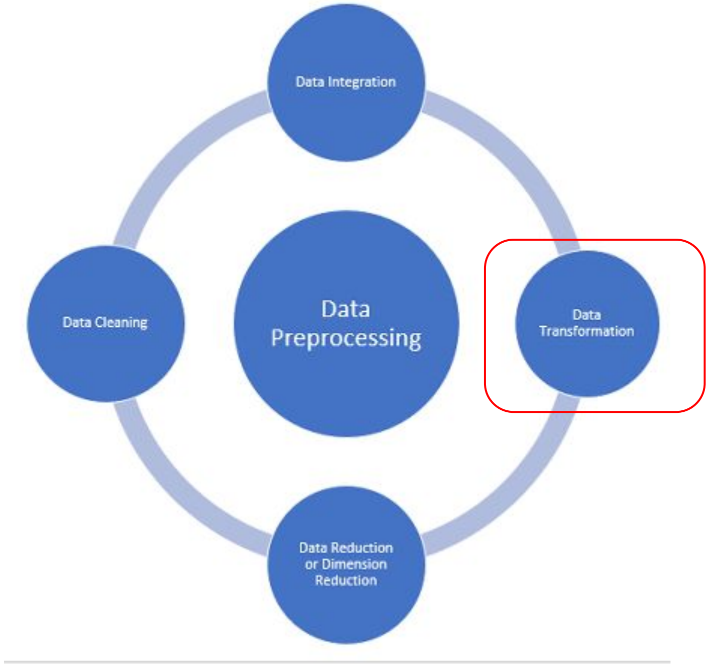
- Background
- UPLIFT System Architecture
- FTBench Benchmark
- Experiments

Feature Transformations

ML Life Cycle Steps:



Data Preprocessing Steps:



Feature Transformations

- **Common Feature Transformations:**
  - **Numerical:** Normalization, **Binning (Bin)**, Aggregation, Scaling
  - **Categorical Transformations:** **Recoding (RC)**, **Dummy-coding (DC)** (one-hot encoding), and **Feature Hashing (FH)**
  - **Modality-specific Transformations:**
    - **Texts:** Bag of words, Word embeddings
    - **Images:** Cropping, Rotating, Contrast adjusting

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

Feature Transformations

- **Binning (Bin)** converts continuous or numerical data into categorical ex.(18-30 -> Young)
  - **Equi-width binning (BinW)** - equal range
  - **Equi-height binning (BinH)** - equal amount
- **Recoding (RC)** modifying the values of a variable to create a new representation that better aligns with the requirements ex.(Young -> 1, Middle-aged -> 2)
- **Dummy-coding (DC)** represent categorical variables as binary or "dummy" variables in ml ex.(Color -> "IsRed":1, "IsBlue":0 , "IsGreen":0)
- **Feature Hashing (FH)** applies hash function to each feature, which maps the original feature values to a fixed number of hash buckets or indices

Challenges of Feature Transformations

1. Large number of **output columns**
2. Many **distinct items per column** (up to millions)
3. **Sparsity and cardinality skew** (proportion of zero or empty values) (tens to millions)
4. **Expensive string processing** (ex. hashing and parsing)
5. **Ultra-sparse outputs** (ex. dummy-coding)
6. **Larger-than-memory output data** (e.g., due to replicated embeddings)
7. **Wide diversity of transformations**
   a. Feature Engineering to find the best combination of FT

Existing Approaches

- Caching and reuse of pre-processing operations
- Interleaving element-wise transformations with data loading
- Static parallelism (row/column-wise)

Good runtime for simple transformations but are **suboptimal for complex, multi-pass transformation workflows**, and **challenging data characteristics** (many features/distinct items).

UPLIFT System Architecture

- UPLIFT creates and optimizes fine-grained task graphs
- Rule-based Optimizer
  - Rewrites according to data, hardware, and operation characteristics
  - Increase fine-grained parallelism by row partitioning
- Cache-conscious Runtime Techniques
- Integrated in Apache SystemDS

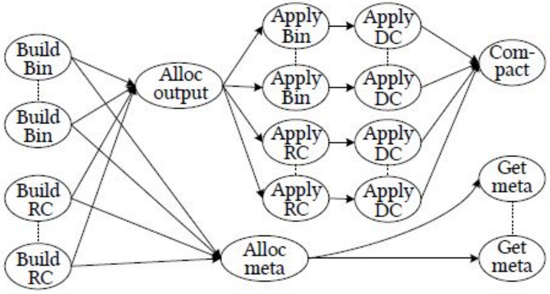


Figure 1: Task Graph for Adult Dataset.

Apache SystemDS™

Task-graph Construction

UPLIFT reads the transformation specification(JSON configuration) as input and create general and encoder-specific tasks

Task Types:

1. **Build** - scans an assigned feature of the input data frame and creates the necessary metadata
2. **Output Allocation -** creates and allocates the output matrix
3. **Metadata Allocation -** creates and allocates a frame for materializing all encoder's meta data

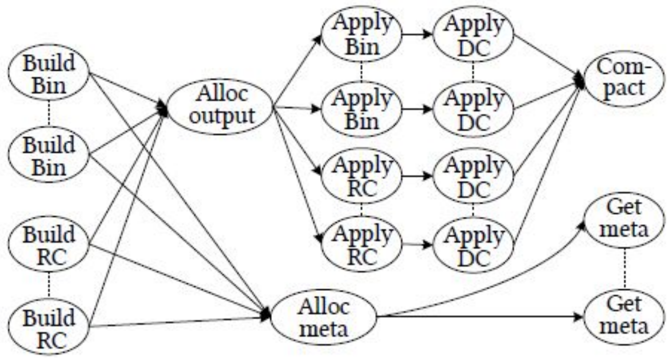


Figure 1: Task Graph for Adult Dataset.

Task-graph Construction

4. **Apply -** reads a feature from the input frame, encodes it using the metadata, and writes the encoded values into the output matrix

5. **Sparse Row Compaction** - compacts sparse rows in-place by removing the zeros (Missing values), shifting the non-zero entries, and updating offsets

6. **Metadata Collection** - serializes the metadata into a frame

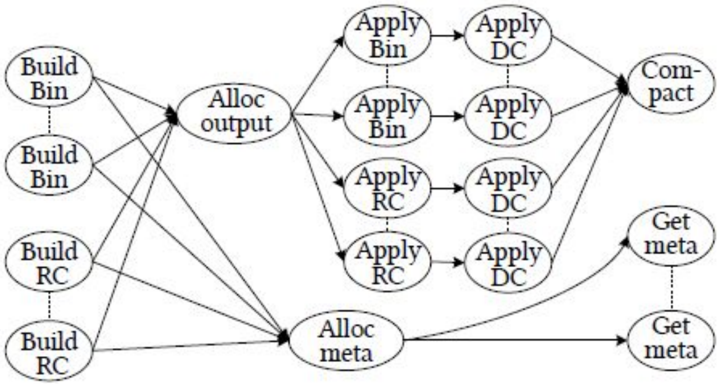


Figure 1: Task Graph for Adult Dataset.

Task-graph Construction

- Create Metadata : ex. Distinct items, bin boundaries
- Pre-allocate output and metadata frame
- Allows concurrent writes and metadata collection
- For CSR (Compressed Sparse Rows) matrix pre-fill row pointers
- Encode input using metadata
- Compacts sparse rows by removing zeros
- Metadata Collection

Rule-based Optimizer

- Reduce Bottlenecks
  - remove unnecessary synchronization barriers, concurrent build, output dimensions are known prior to the build tasks
- Row Partitioning
  - additionally partition a column into multiple row-ranges and assign a task to each block of rows
- Number of Partitions
  - increasing the number of row partitions (tasks operating on row ranges) increases memory overhead
  - finds a good number of partitions for each feature
  - reduce the degree of parallelism if the total memory estimate exceeds the memory budget.

Example of Optimized Task Graphs

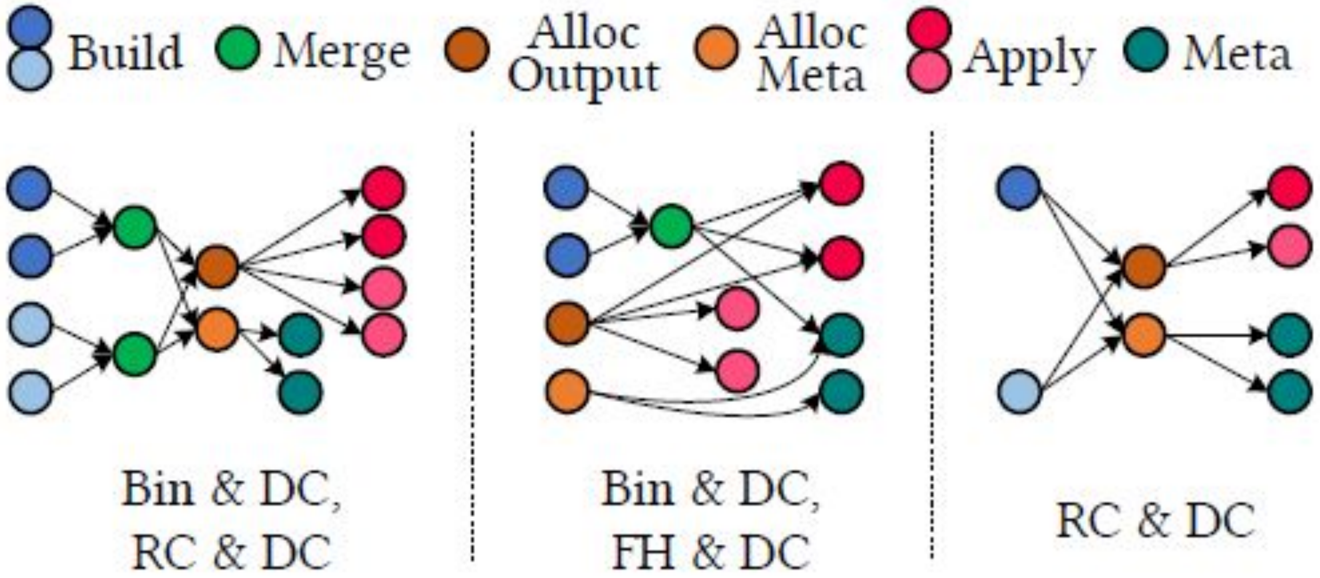


Figure 2: Three Examples of Optimized Task Graphs.

Feature Transformation Benchmark - FTBench

Foster Research on Feature Transformations

- Datasets
  - Publicly available and synthetic datasets
  - Sources: UCI, Kaggle, AMiner
  - Datasets to capture choke points (previously reported challenges)
- Use Cases
  - Domains and modalities (numerical, categorical, text, and time series)
  - Data and transformation characteristics (#distincts, distribution of distinct values, #bins, string lengths, and sparsity)
  - Workload types (batch and mini-batch)
  - Scale factors for selected use cases

Feature Transformation Benchmark - FTBench

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

Feature Transformation Benchmark - FTBench

Table 2: Overview of FTBench Datasets and Use Cases.

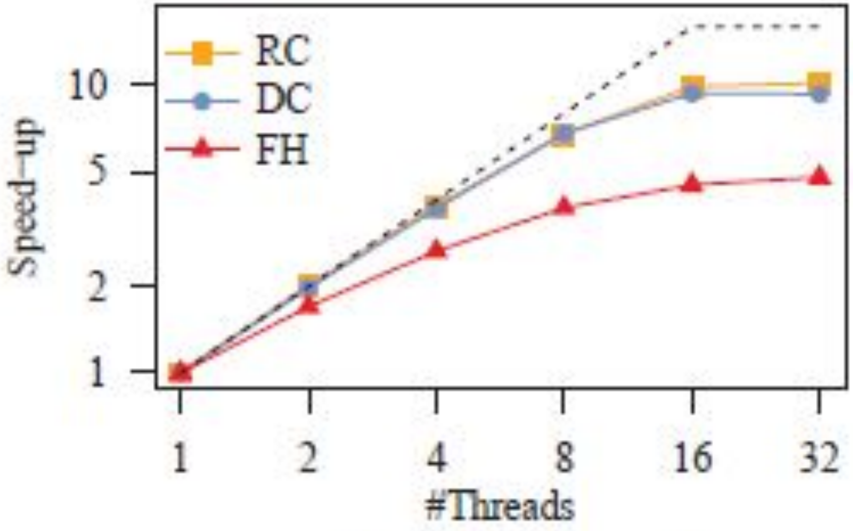
ID	Dataset	Input Shape	Transformations	Significance	Output Shape
T1	Adult	32K × 15	Bin+DC (5), DC (9), PT (1)	Popular dataset	32K × 130
T2	KDD 98	95K × 469	Bin (334), DC (135), Scale (469)	Skewed #distinct: 50-900	95K × 6K
T3	Criteo	10M × 39	DC (26)	Skewed & large #distinct: 10-1.4M	10M × 5.8M
T4	Criteo	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

Experimental Setting

- **Hardware/Software:** Ubuntu 20.04.1, single AMD EPYC 7302 CPU @3.0-3.3 GHz (16 physical/ 32 virtual cores), OpenJDK 11, Python 3.8
- Compare **UPLIFT** with **Apache SystemDS (Base), SKlearn, other** (Spark, Dask, Keras, Tensorflow)
- **Datasets:** FTBench benchmark

Micro Benchmarks

- **Speedup of UPLIFT with increasing #threads**



(a) Speedup w/ #Threads

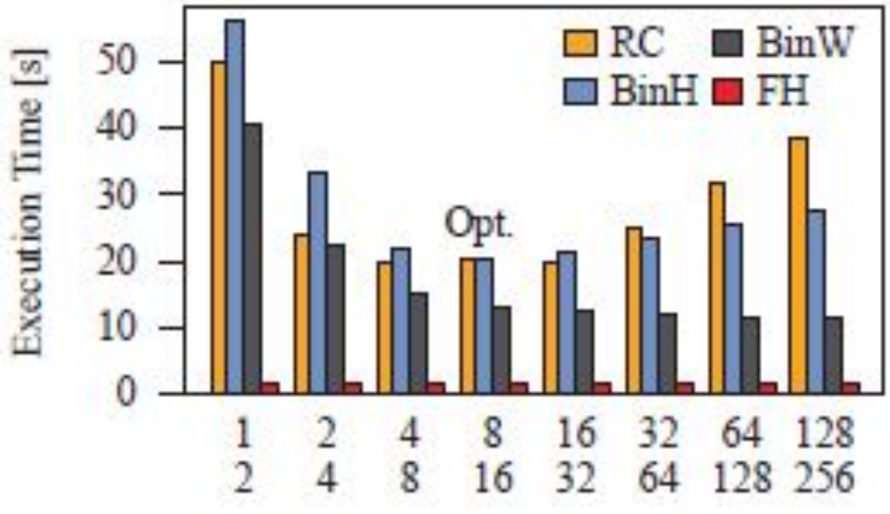
Dataset: 5M x 100 (100K #distinct each)
Transformations

RC = Recoding
DC = Dummy coding
FH = Feature hash (k = 10K)

- RC improves up to **10x at 16 physical cores**
- DC produces **10M columns** (ultra-sparse) but equally well
- FH smaller bc memory-bandwidth bound

Micro Benchmarks

- **Impact of partitions**



(d) #Build/#Apply Partitions

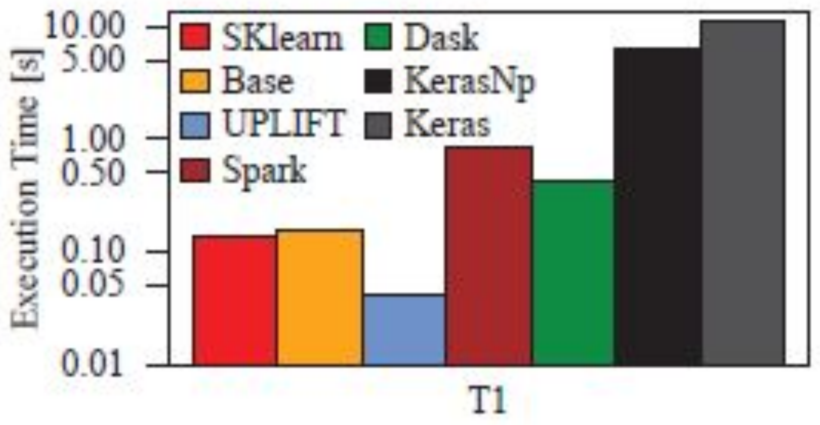
Dataset: 100M x 4 (1M #distinct each)

Transformations:
RC = Recoding
DC = Dummy coding
BinW = Equi-width binning
BinH = Equi-height binning
FH = Feature Hash

- **Performance improves up to 8/16 partitions**
- FH is robust to partitioning (no metadata)
- **UPLIFT optimizer also picks 8/16**

FTBench Implementations

- **Small dataset (T1 Adult Dataset)**



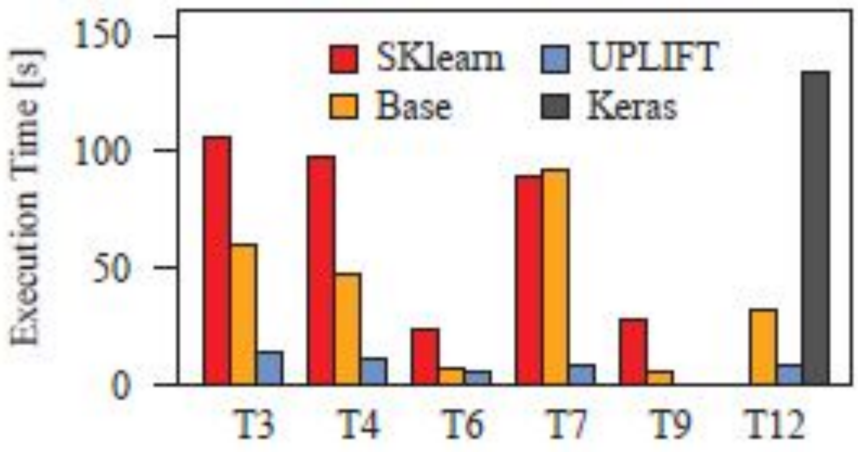
(a) Adult Dataset

Baselines:
Base = SystemDS default config
KerasNp = Keras build w/ Numpy. unique

- Base, SKlearn are **32x/52x** faster than Keras
- UPLIFT further improves by 6x
- Dask, Spark's **static parallelization schemes are ineffective for smaller datasets**
- UPLIFT is **10x faster** than Spark.ml

FTBench Implementations

- **Large Datasets**

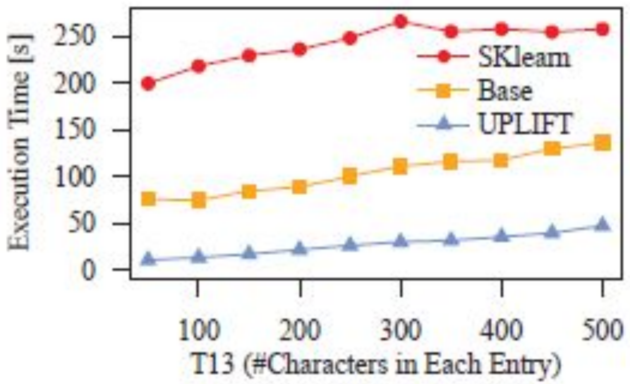


(c) Large Datasets

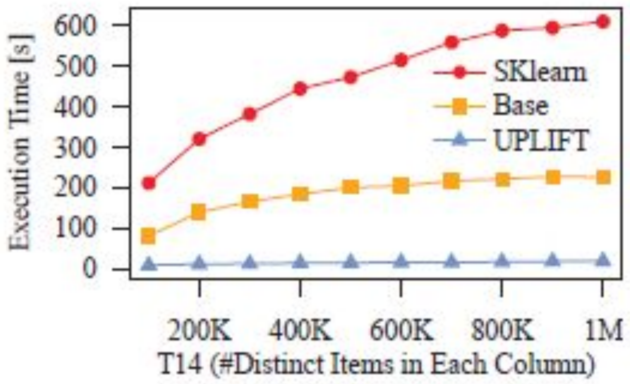
- UPLIFT is consistently **faster** than Base and Sklearn
- On Criteo(T3) Spark is 2.5x faster than Sklearn
- For T3, UPLIFT is 3x faster than Spark
- **Dynamic parallelization schemes significantly improve across different data characteristics**

FTBench Implementations

- **Varying Data Characteristics**



(f) String Length



(g) Distinct Values

Conclusions

- **UPLIFT** as a parallel feature transformation framework with **fine-grained task scheduling**
- **Optimization** based on data, workload and hardware characteristics
- **UPLIFT** showed **good improvements** compared to static parallelization
- During the development of UPLIFT, FTBench already proved to be very useful
- UPLIFT is fully integrated in **Apache SystemDS**

Future:

- ***Runtime Backends:*** Extending UPLIFT to distributed, data-parallel operations, federated backends (learning process occurs across multiple devices or servers). Now only local operations on CPUs
- ***Optimizer Guarantees:*** UPLIFT doesn’t yet provide guarantees on finding cost-optimal plans, or ensuring not to exceed the given memory budget
- ***Implementations*** for more baseline ML systems