SCIENCE
PASSION
TECHNOLOGY

UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads

Arnab Phani[1], Lukas Erlbacher[1], Matthias Boehm[1]

[1] **Graz University of Technology**; Graz, Austria

Feature Transformations

- **Transform Raw Data into Numeric Representation**
  - Part of **feature engineering**
  - Scales, modifies, or converts features
  - Placed **before or within** (batch-wise) training pipeline
- **Common Feature Transformations**
  - Numerical: aggregations, scaling, binning
  - Categorical: recoding, dummy coding, feature hashing
- **Modality-specific Feature Transformations**
  - Texts: Bag of words, embedding
  - Images: cropping, rotating, adjusting contrast

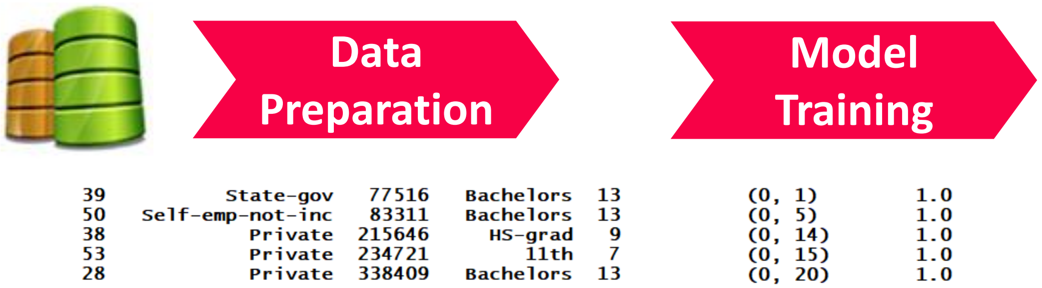
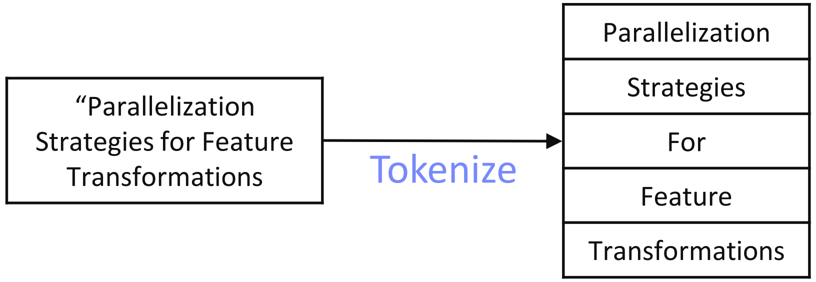
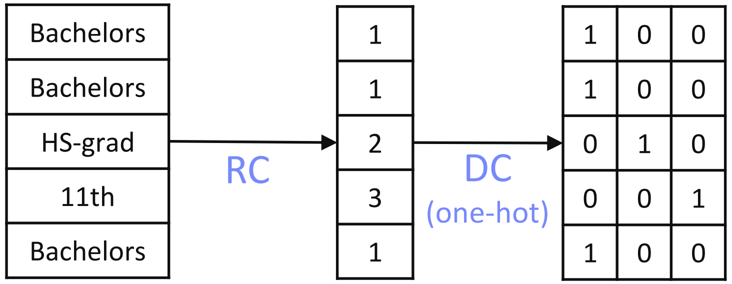


Table 1: Common Multi-pass Transformations.

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





Challenges of Feature Transformations

- **Multi-pass Nature (Build, Apply)**
- **Many Distinct Items per Column**
  - Columns with **10M to 20M distinct values**
- **Sparsity, Cardinality Skew (#distincts from tens to millions)**
- **Expensive String Processing (hashing and parsing)**
- **Ultra-sparse Outputs (e.g. one-hot encoding)**
- **Wide Diversity of Transformations**
  - **Feature engineering** to find the best combination
  - Accuracy varies drastically with feature transformations

- **Existing Approaches**
  - Caching and **reuse** of pre-processing operations
  - Interleave element-wise transformations with data loading
  - **Static parallelism** (row-/column-wise)
  - Suboptimal for complex transformation workflows

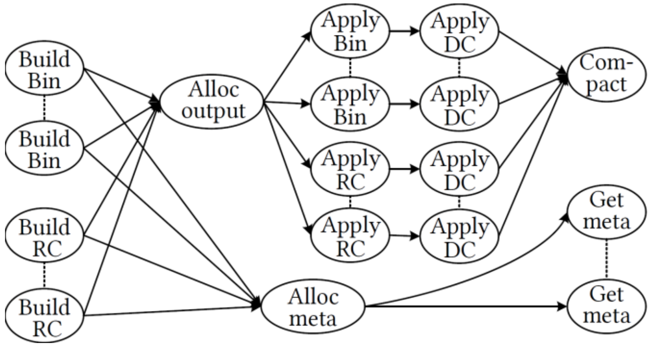
[Criteo AI Lab. 2020. Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/]

[Doris Xin et al: Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. **SIGMOD '21**]

This multi-pass nature and potentially many or large dictionaries make simple, **data-parallel execution ineffective**.

UPLIFT and FTBench

- **Exploiting Available Parallelism**
  - Construct **fine-grained task-dependency graph** (execution plan)
- **Rule-based Optimizer**
  - Rewrites according to data, hardware, and operation characteristics
  - Remove synchronization barriers
  - Increase fine-grained parallelism by **row partitioning**
- **Cache-conscious Runtime Techniques**
- **FTBench, Feature Transformation Benchmark**
  - **Foster research** on feature transformations
  - Use publicly available **real and synthetic datasets**
  - Covers different domains, modalities, transformations, data characteristics, and workload types
- **Multiple Reference Implementations**
  - Full Implementations: UPLIFT/SystemDS, scikit-learn
  - Partial Implementations: TensorFlow Keras, PySpark, Dask,



Integrated into **Apache SystemDS**

Apache SystemDS™

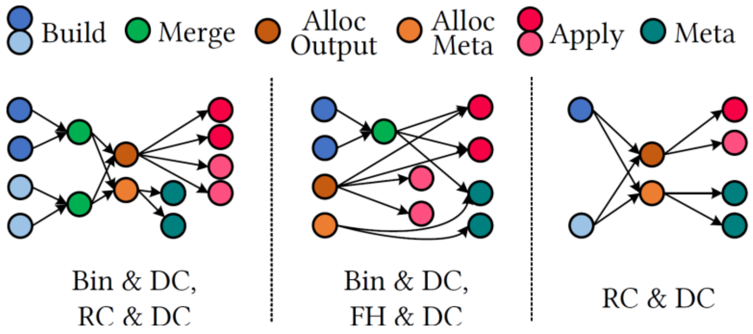
Table 2: Overview of FTBench Datasets and Use Cases.

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

Apache SystemDS™ | scikit learn | Apache Spark ML | TensorFlow | DASK

UPLIFT System Architecture

(Task-graph Construction and Optimizations)



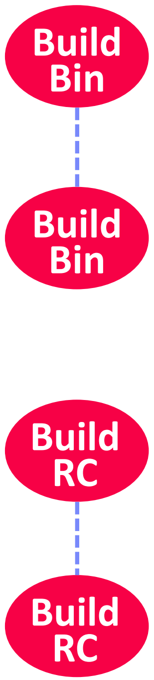
Task-graph Construction

- **Create General and Encoder-specific Tasks**
- **Adult dataset with 6 numerical and 9 categorical features**

```
[X_enc, Meta] =
    transformencode(target=data, spec=jspec);
```

Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols

Build Tasks



- Column- oriented
- **Create metadata**. E.g. Distinct items, bin boundaries

Input Frame

Bachelors
Bachelors
HS-grad
11th
Bachelors

Metadata

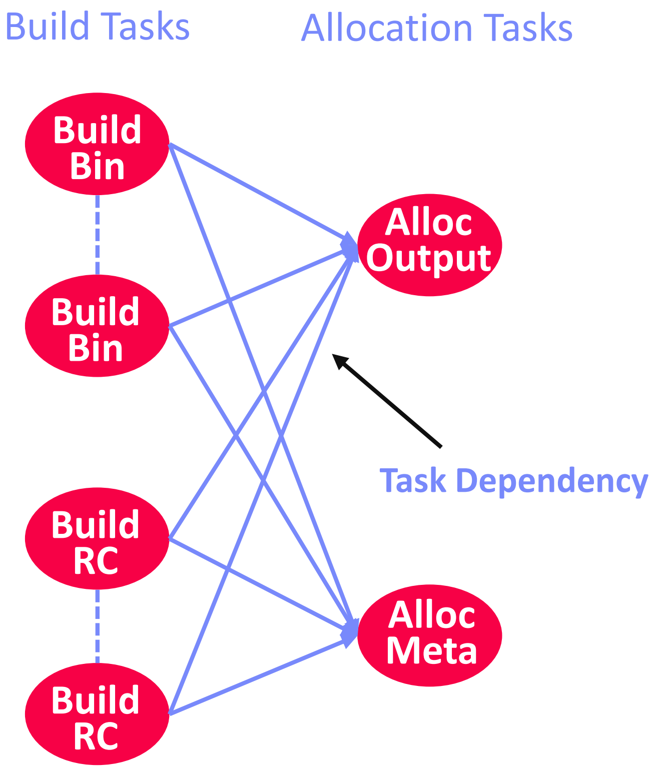
Bachelors	1
HS-grad	2
11th	3

Task-graph Construction

- **Create General and Encoder-specific Tasks**
- **Adult dataset with 6 numerical and 9 categorical features**

```
[X_enc, Meta] =
    transformencode(target=data, spec=jspec);
```

Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols



- Determine **upper bound of #non-zeros**
- **Pre-allocate** output (dense/sparse) and metadata frame
- Allows **concurrent writes** and metadata collection
- For CSR, pre-fill row pointers

Input Frame

Bachelors	US
Bachelors	US
HS-grad	UK
11th	US
Bachelors	IN

CSR Output Allocation

#nonZeros = r * c = 10
#output_rows = r = 5
#output_cols = 3 + 3 = 6

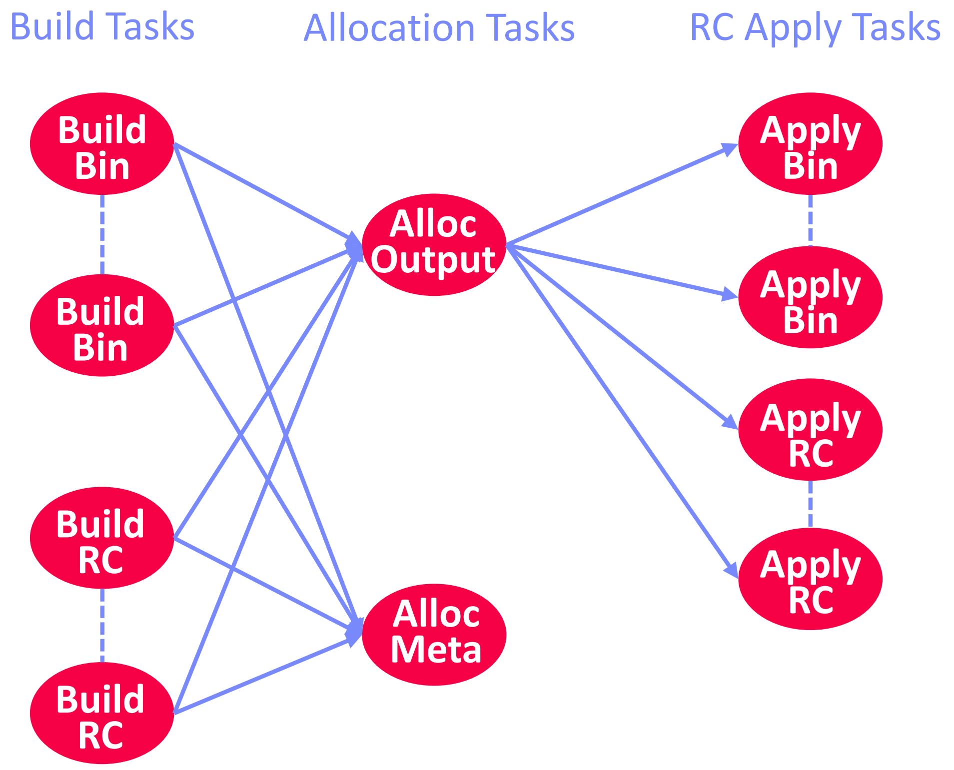
V = [. . .] #nnz
Col_Index = [. . .] #nnz
Row_Index = [0, 2, 4, 6, 8]

Task-graph Construction

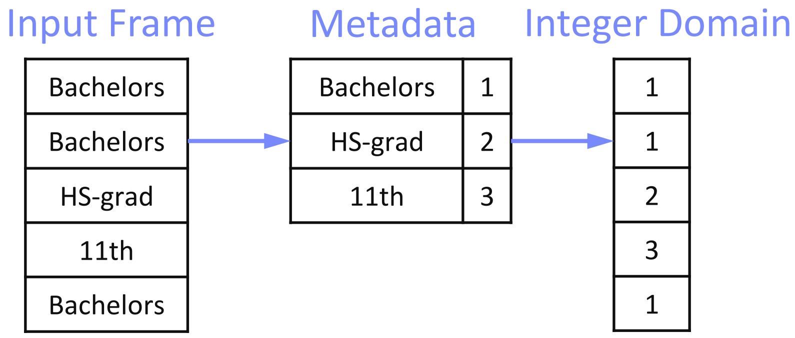
- **Create General and Encoder-specific Tasks**
- **Adult dataset with 6 numerical and 9 categorical features**

```
[X_enc, Meta] =
    transformencode(target=data, spec=jspec);
```

Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols



- Column-oriented
- Encode the input using the **metadata**
- Produce **continuous integer domains**

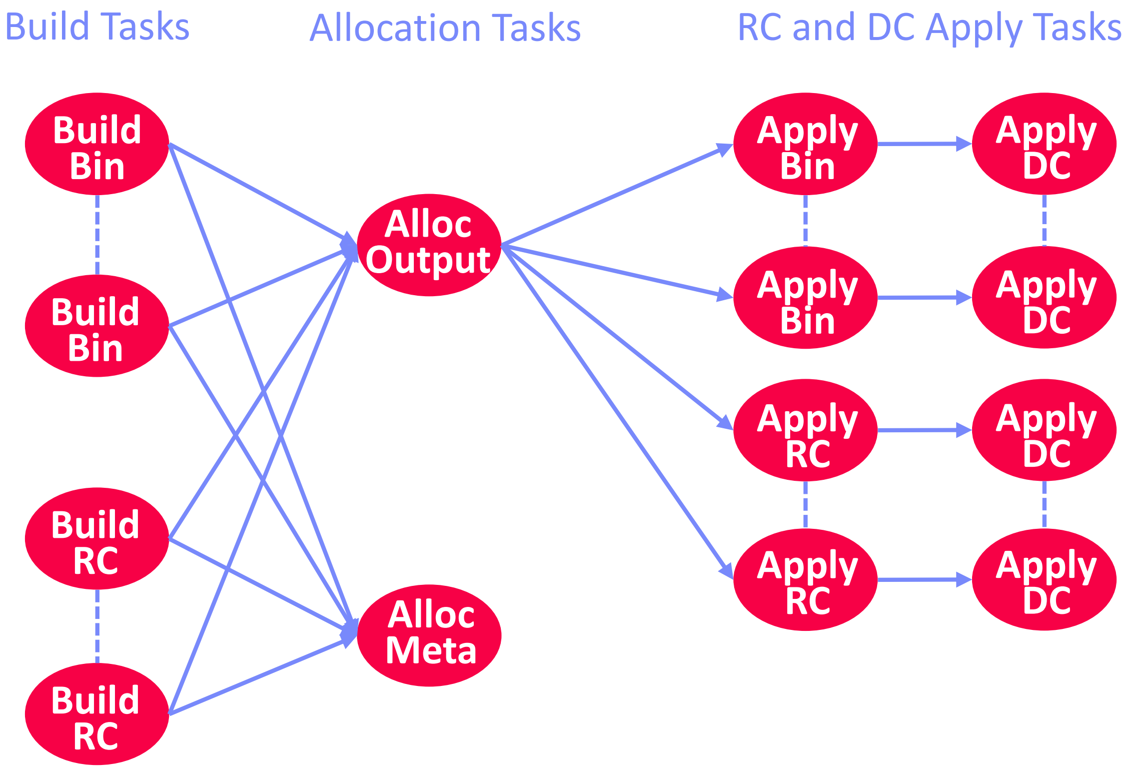


Task-graph Construction

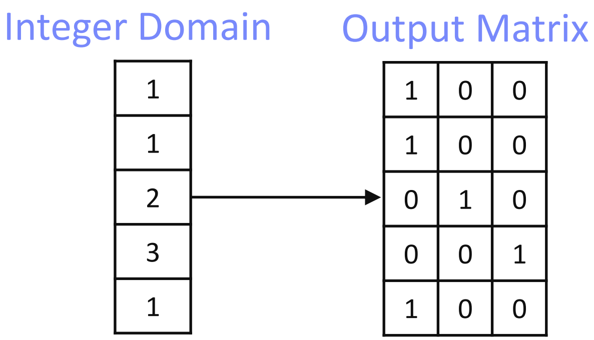
- **Create General and Encoder-specific Tasks**
- **Adult dataset with 6 numerical and 9 categorical features**

```
[X_enc, Meta] =
    transformencode(target=data, spec=jspec);
```

Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols



- Column-oriented
- Integer domains to sparse binary vectors
- **Ultra-sparse output** with large #columns

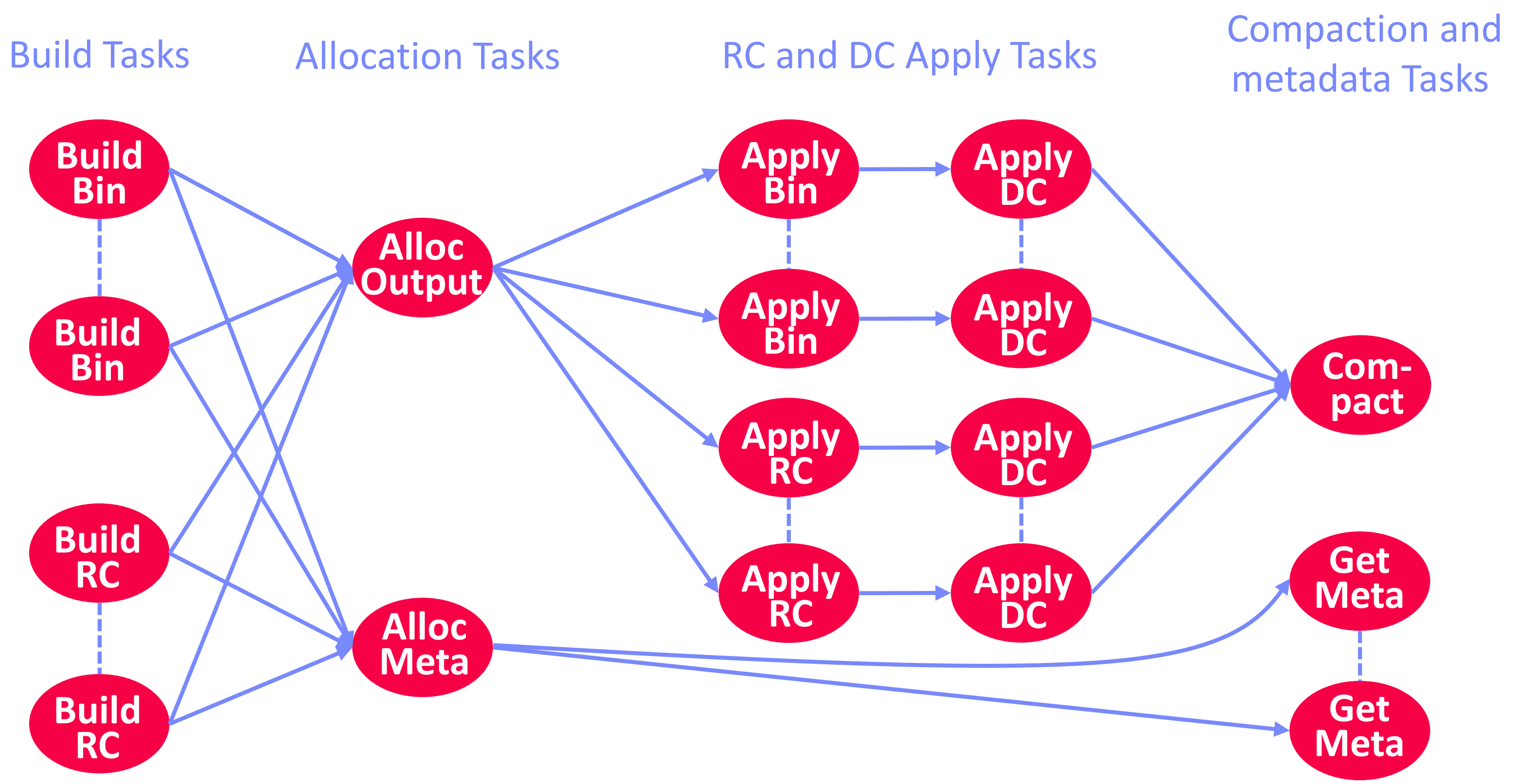


Task-graph Construction

- **Create General and Encoder-specific Tasks**
- **Adult dataset with 6 numerical and 9 categorical features**

```
[X_enc, Meta] =
    transformencode(target=data, spec=jspec);
```

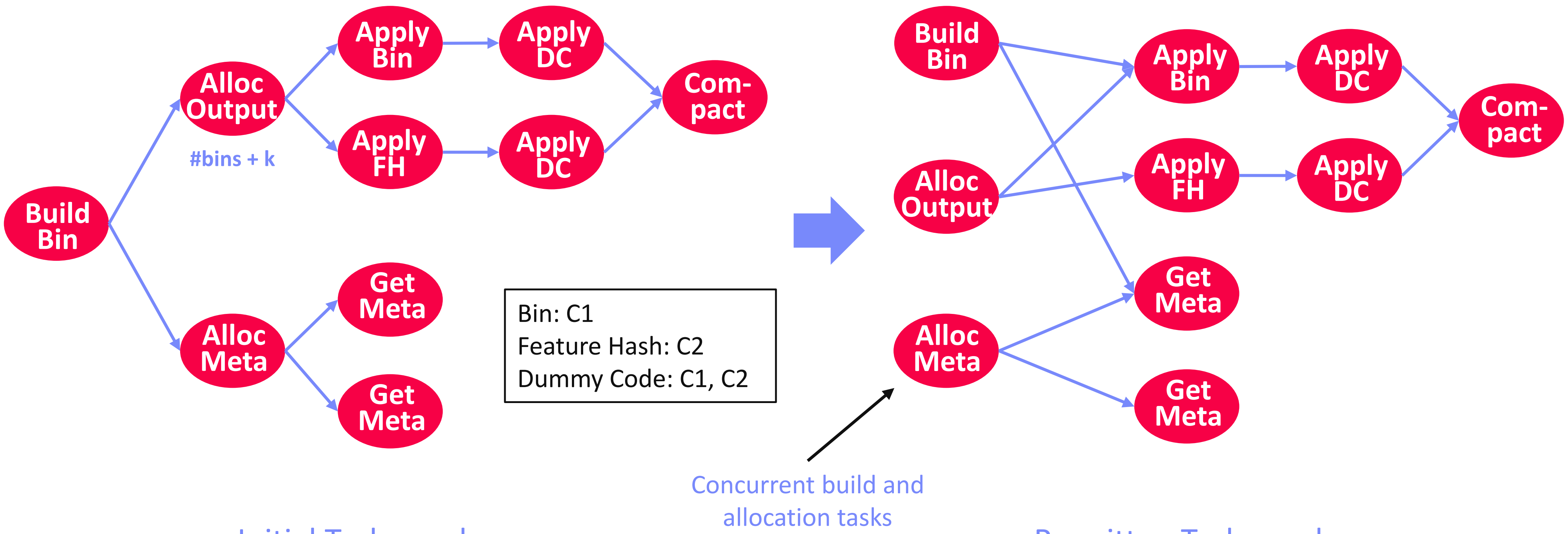
Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols



- Row-parallel
- Compacts sparse rows **in-place** by removing zeros
- **Serializes metadata** in column-wise manner

Optimizer Rewrites

- **Remove Synchronization Bottlenecks**
  - E.g. concurrent build and allocation tasks if output dimensions are known

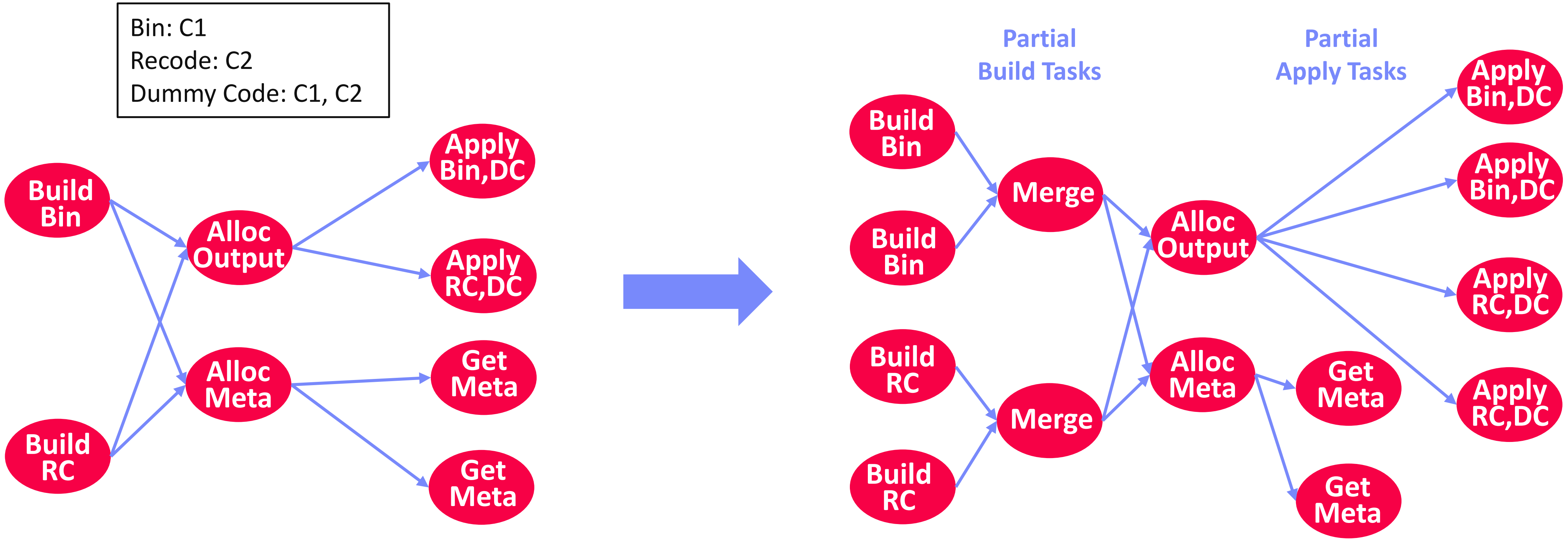


Initial Task-graph

Rewritten Task-graph

Optimizer Rewrites

- **Row Partitioning**
  - Column-oriented tasks fails to fully utilize all cores if **#features < #cores** or **skewed compute time**
  - Partition columns into **row-ranges** and assign a task to each block of rows



Initial Task-graph

Rewritten Task-graph

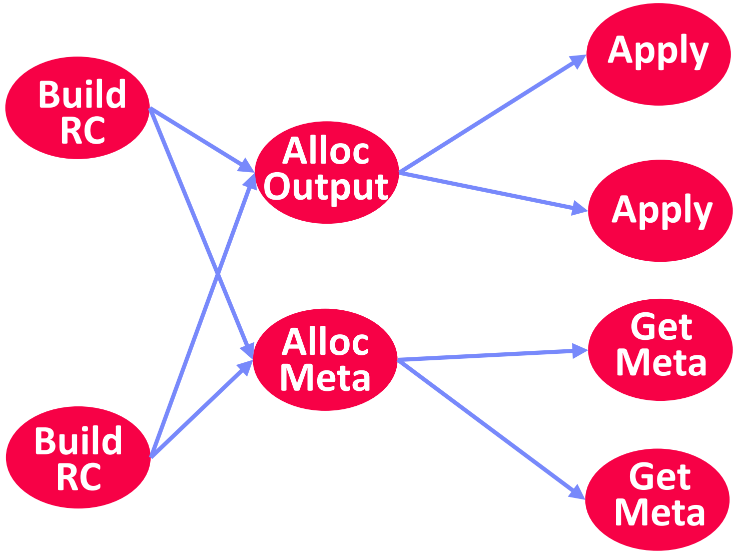
Number of Partitions

- **Increasing #Partitions Increases Memory Overhead**
- **Estimate Memory Usage of Partial Tasks**
  - Collect uniform **sample of rows**
  - Estimate **#distinct items** and **average string size**

[Peter J. Haas and Lynne Stokes: Estimating the Number of Classes in a Finite Population. **JASA 1998**]

- **Derive Optimum Number of Partitions**
  - Schedule more tasks than cores to **mitigate skew.** `2 x #cores` for build and `4 x #cores` for apply tasks
  - Tune #partitions to **fit into memory budget**

Recode: C1, C2
Dummy Code: C1, C2

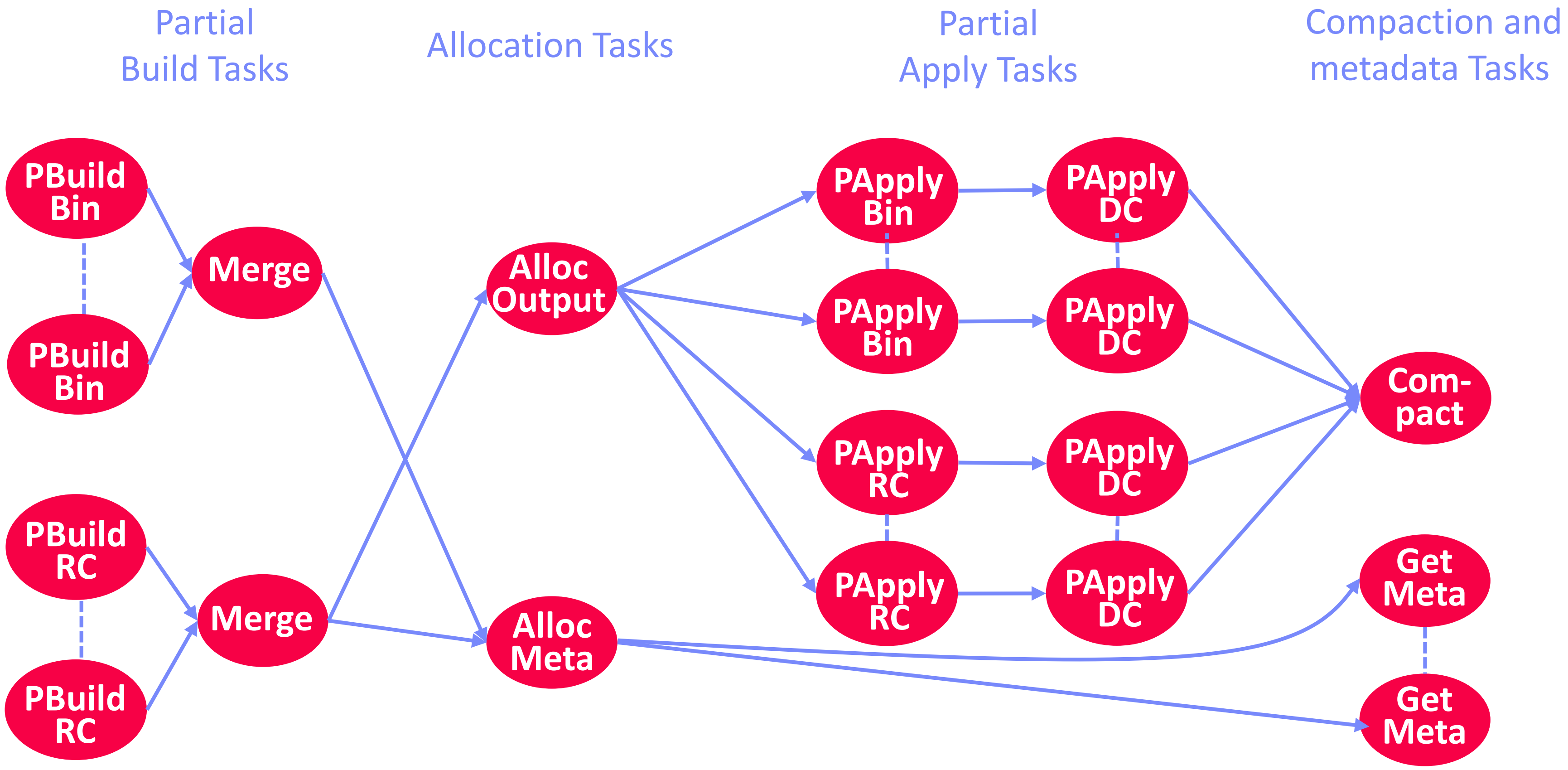


We schedule a single recode build-task per feature because the **estimated total size of the partial maps exceeds memory constraint**.

Optimized Task-graph for Adult

- **Remove Dependency between Bin Build and Alloc Output**
- **Partition Build and Apply Tasks**

Bin: 6 num. cols
RC: 9 cat. cols
Dummy Code: 15 cols



Feature Transformation Benchmark

(Datasets and Use Cases)

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

FTBench – Feature Transformation Benchmark

- **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

FTBench – Feature Transformation Benchmark

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

Covering various domains, modalities and transformations

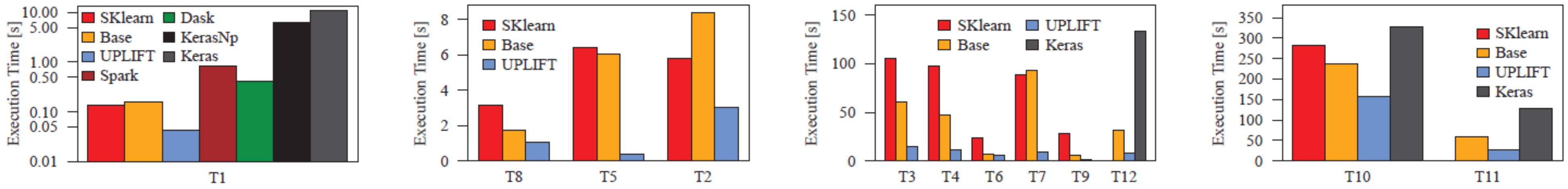
FTBench – Feature Transformation Benchmark

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

Choke points and scaled use cases

Experiments

(Micro Benchmarks and FTBench)



(a) Adult Dataset (b) Small Datasets (c) Large Datasets (d) Text Datasets

Experimental Setting

- **Baselines**

- **Datasets and Workloads**

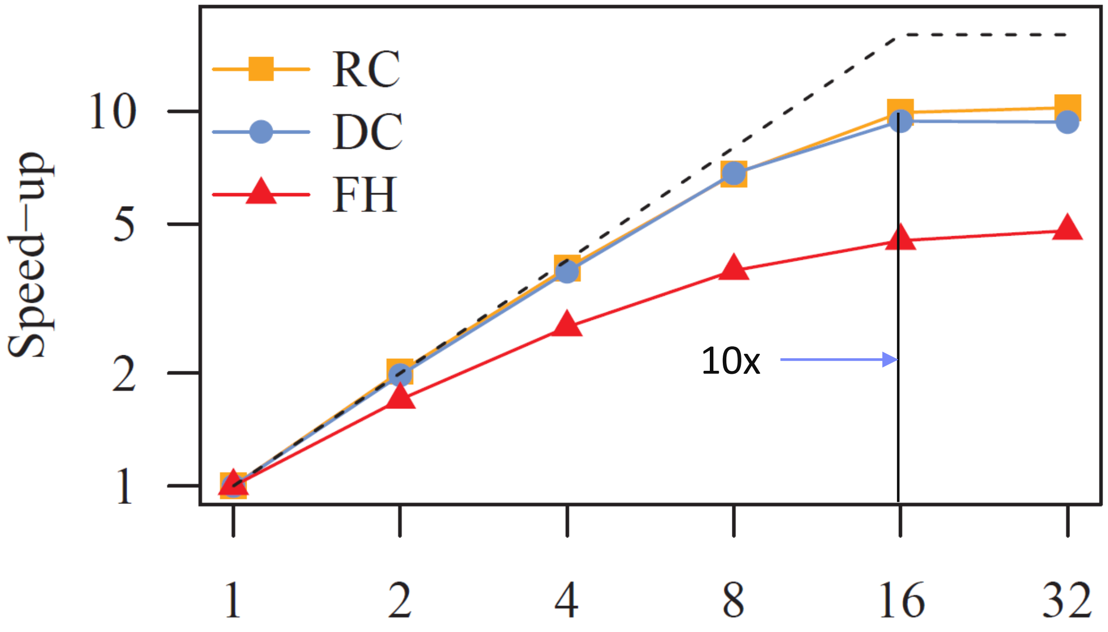
All use cases of **FTBench** benchmark. Two full reference implementations and a few partial implementations.

- **Hardware**

Single node having a single AMD EPYC 7302 CPUs (**16/32 cores**) and **128 GB** DDR4 RAM

Micro Benchmarks

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

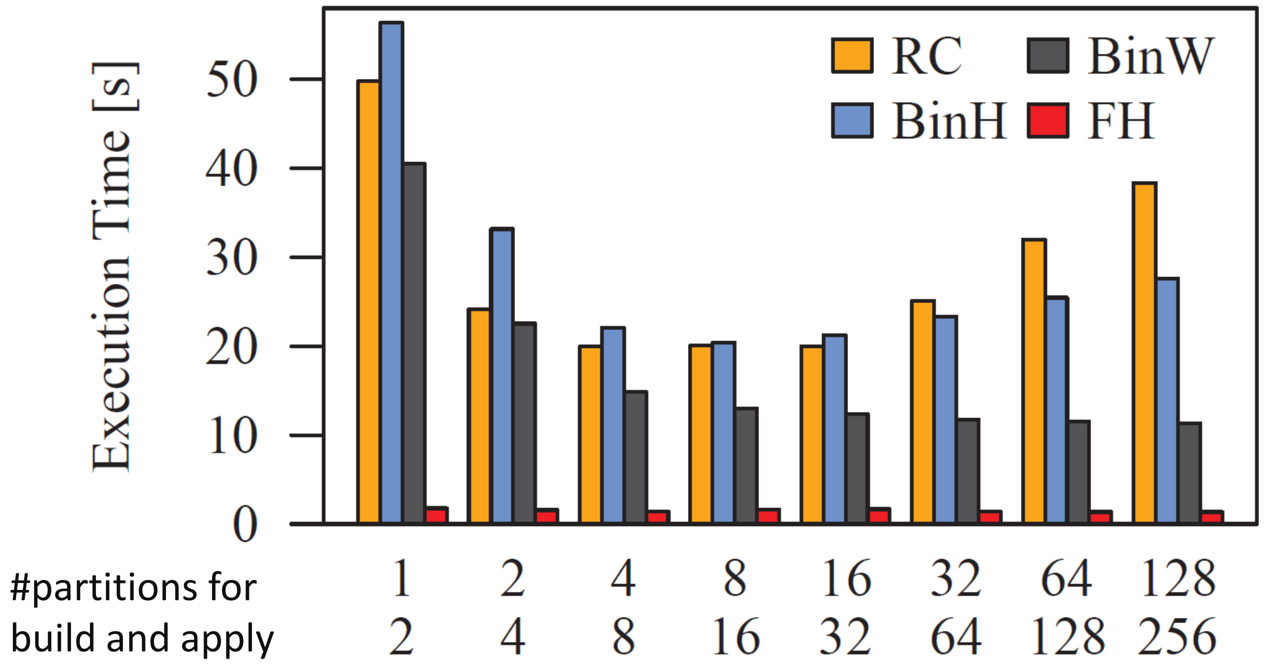


Dataset: 5M x 100 (100K #distinct each)
Transformations:
#1 RC = Recoding
#2 DC = Dummy coding
#3 FH = Feature hash (k = 10K)

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

Micro Benchmarks

- **Impact of #partitions (tasks)**

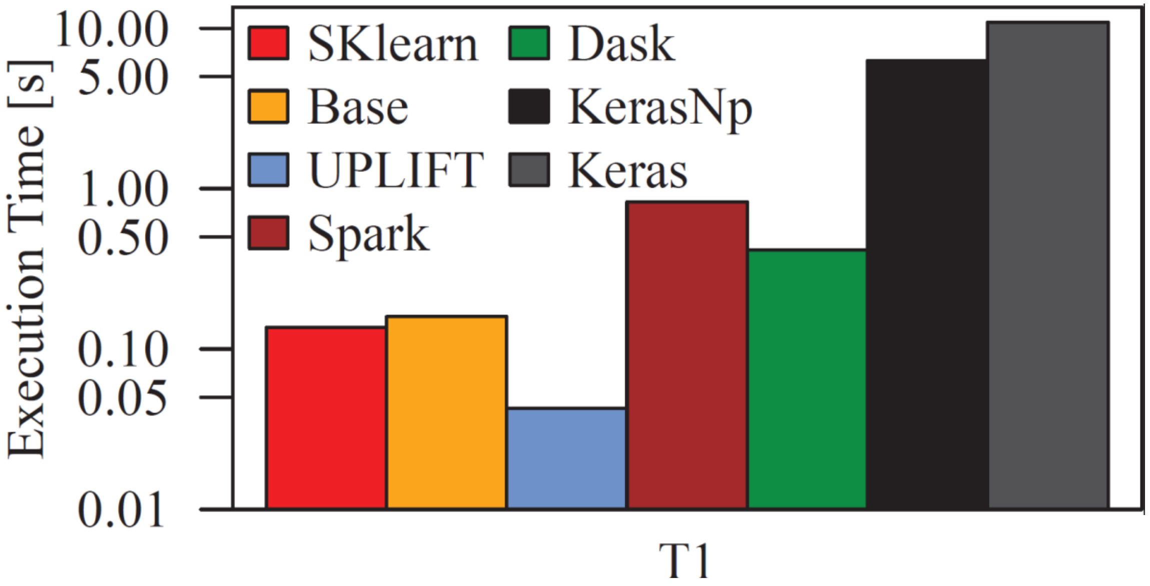


Dataset: 100M x 4 (1M #distinct each)
Transformations:
#1 RC = Recoding
#2 DC = Dummy coding
#3 BinW = Equi-width binning (10)
#4 BinH = Equi-height binning (10)
#5 FH = Feature Hash

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

FTBench Implementations

- **Use Case T1 (Adult Dataset)**



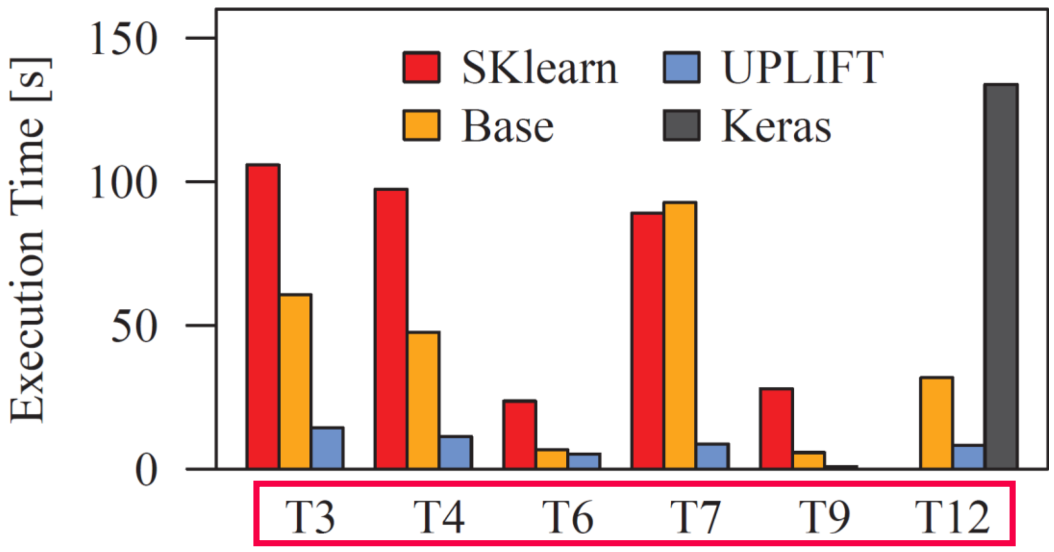
Baselines:
#2 Base = SystemDS default config
#6 KerasNp = Keras build w/ **`Numpy.unique`**
#7 Keras = Keras build w/ **`adapt`**

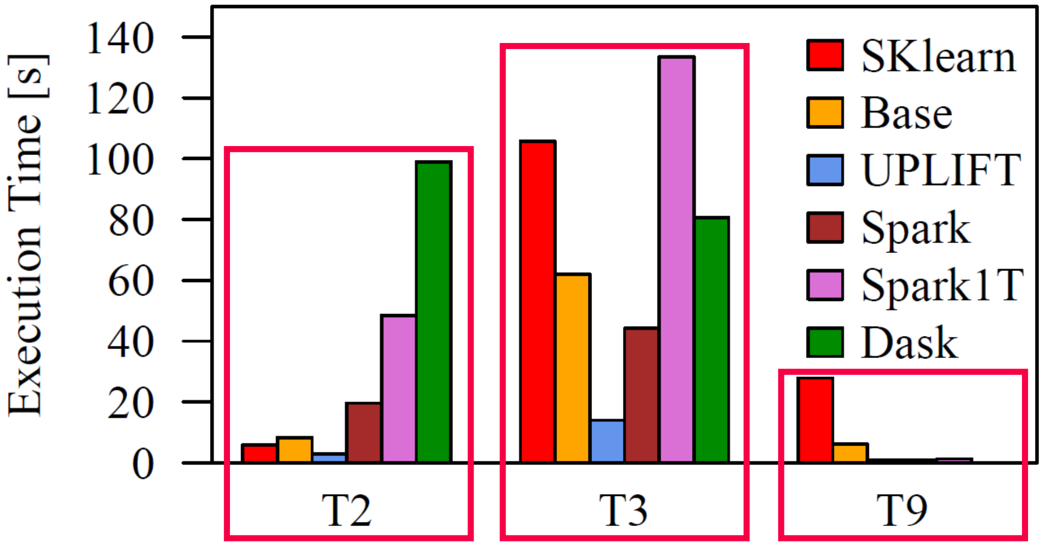
- Base, SKlearn are **32x/52x** faster than Keras
- UPLIFT further improves by **6x**
- Dask, Spark's **static parallelization schemes are ineffective for smaller datasets**
- UPLIFT is **10x** faster than Spark.ml

FTBench Implementations

- **Datasets with Structured Attributes**
  - 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**

Baselines:
#1 SKlearn = Scikit-learn
#2 Base = SystemDS default config
#3 Spark = spark.ml
#4 Spark1T = single-threaded spark.ml
#5 Dask = Dask

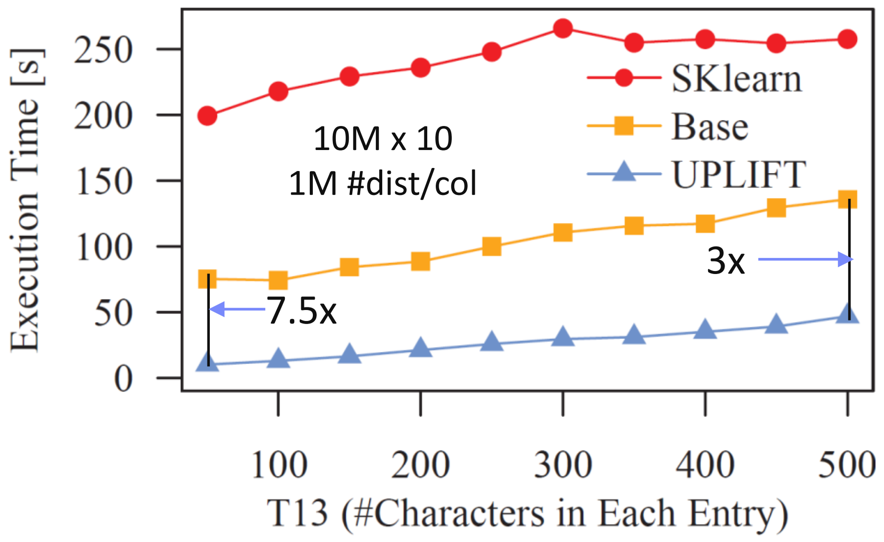




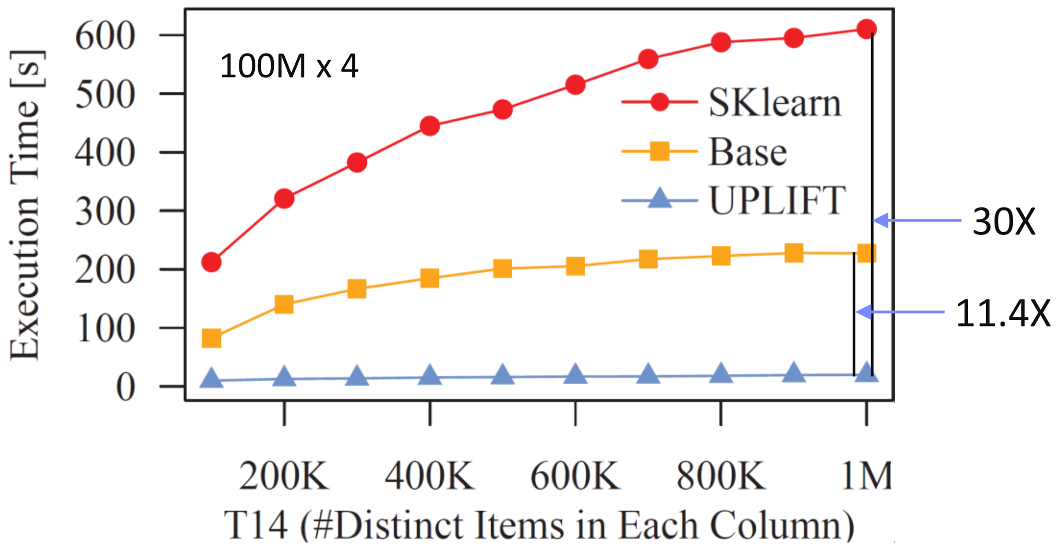
FTBench Implementations

- **Varying Data Characteristics**

Baselines:
#1 Base = SystemDS default config
#2 SKlearn = Scikit-learn



Varying String Length



Varying #Distinct Items

Conclusions

- **Summary**
  - **Parallel** feature transformation
  - **Fine-grained task scheduling** and cache-conscious runtime
  - **Optimization** with regard to data, workload and hardware characteristics
  - FTBench for **evaluating feature transformation frameworks**

The reference implementations are available at https://github.com/damslab/reproducibility/tree/master/vldb2022-UPLIFT-p2528

- **Conclusions**
  - Feature transformations are an incredibly **common part of ML dev**
  - Increasing **multi-modality** makes transformations more challenging
  - UPLIFT's parallelization strategies **proved effective** and foster future research
- **Future Work**
  - Extend to distributed, data-parallel operations, federated backends and hardware accelerators
  - Cost-based optimizer, scan sharing and fusion among transformations
  - **More reference implementations** of FTBench