PandasUDFs – One Weird Trick to Scaled Ensembles

Paul Anzel

Data Engineer – H-E-B

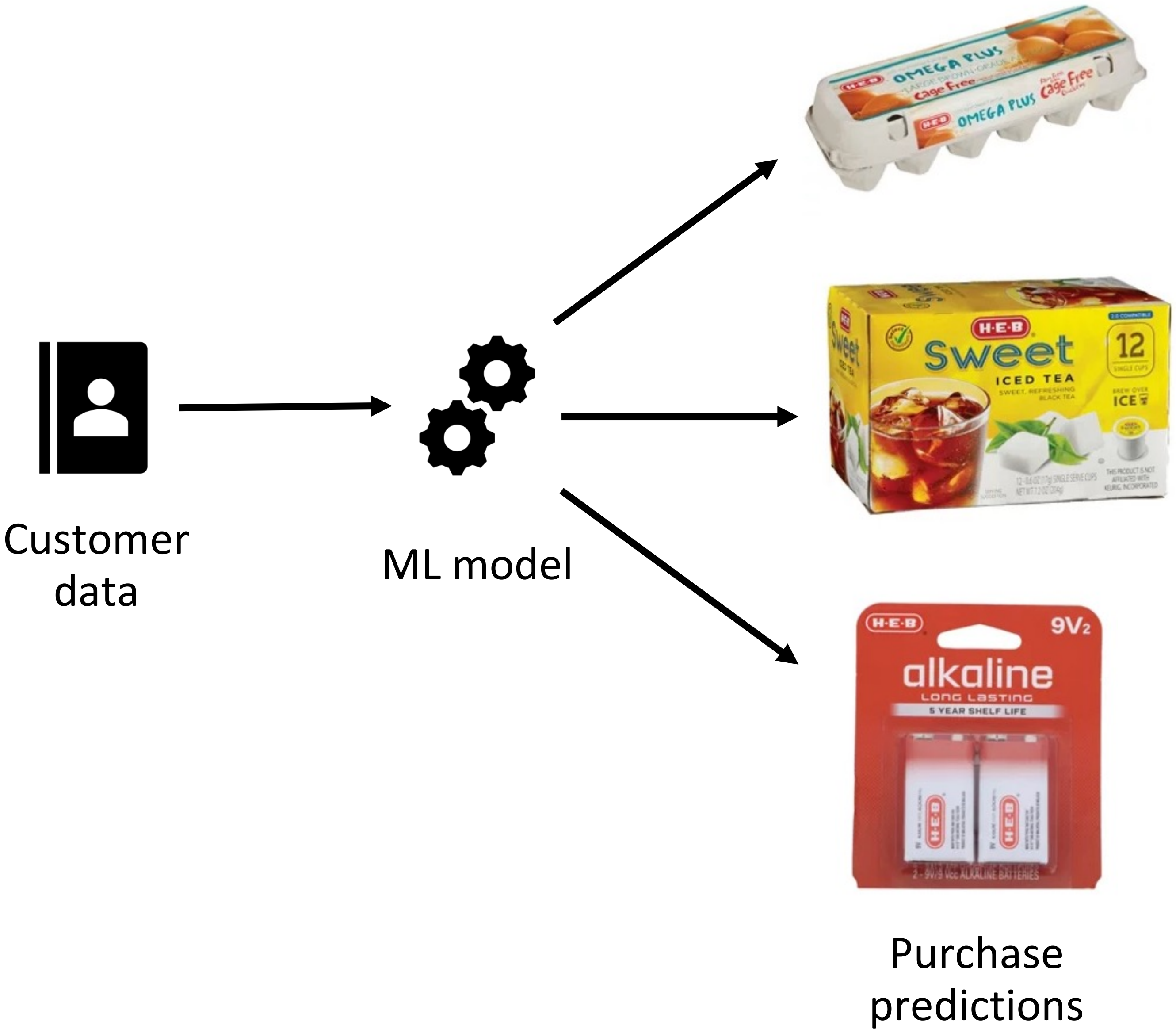
Agenda

- Introduction
- How do we use PandasUDFs?
- The 2 GB limit
- R and Koalas

Introduction

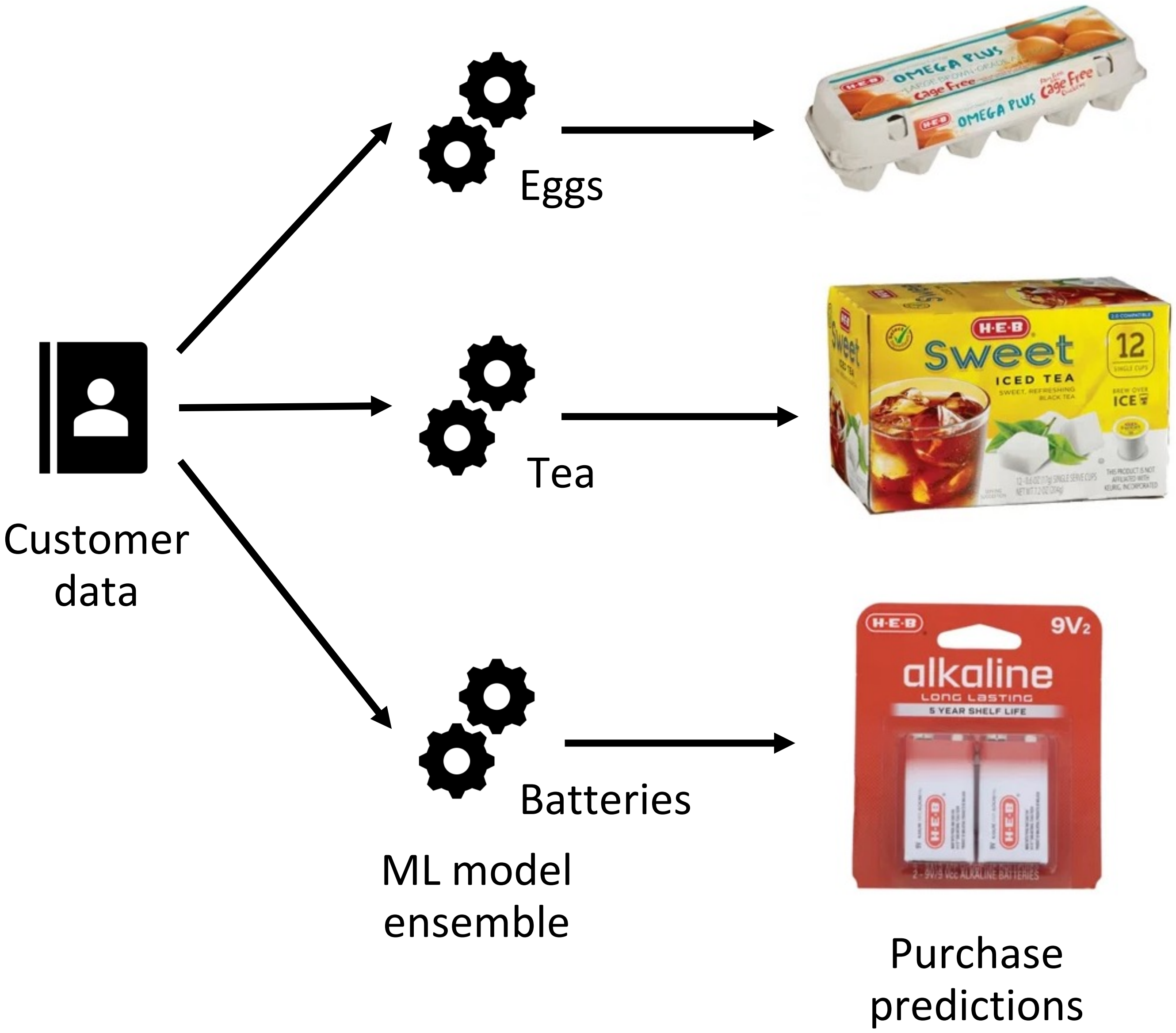
The problem

- Needed to improve model to predicts if customers will purchase in a category.
- Old system – one model for everything!
  - Do you purchase batteries like you purchase eggs?



A solution

- Don’t have one model do every category, make an ensemble of models.
- Dramatically improved metrics, especially for low velocity items.
- Headache to manage and massive runtime.



Here’s the problem

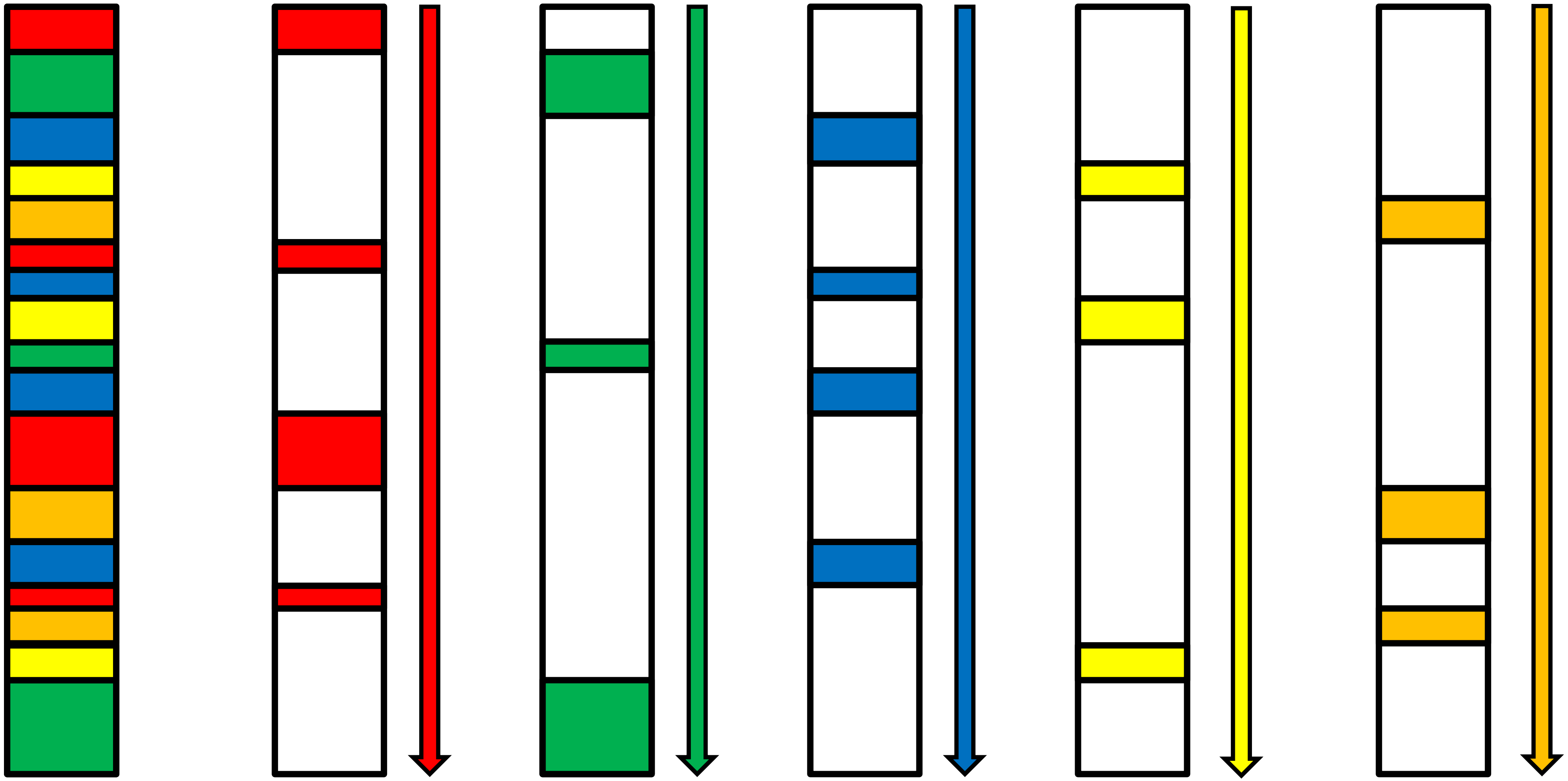
```python
sdf = ...  # Big Spark DataFrame of data

categories = [i.cat for i in sdf.select('id_cat').distinct().collect()]

for category in categories:
    sdf_filtered = sdf.filter(col('id_cat') == category)
    # Do ML process
    ...
```

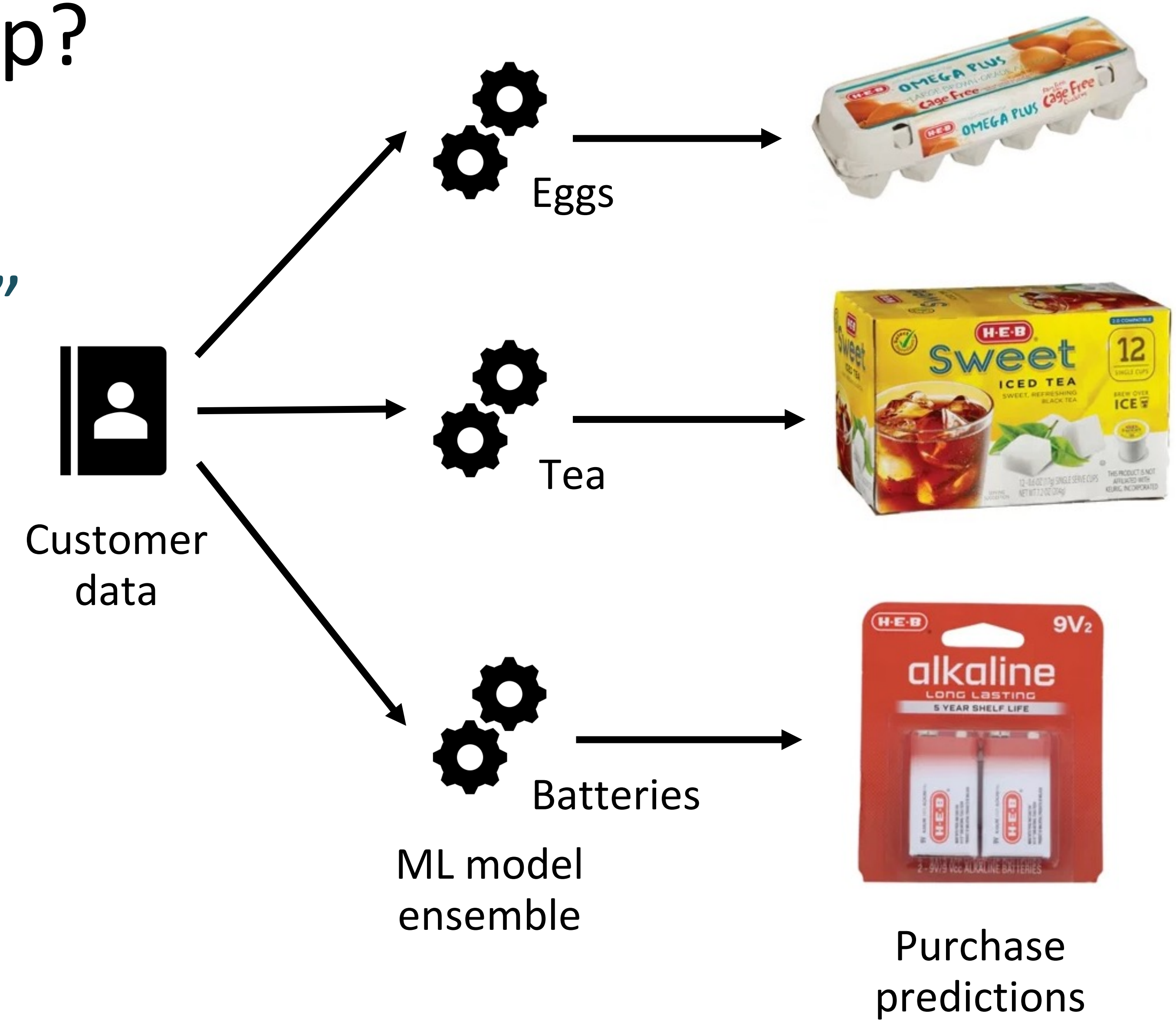
This took >10 hours

What happens



How do we speed this up?

- Training each model is "embarrassingly parallel"
- Initial attempt was via multithreading
  - Processes stepped on each other's toes
  - Doesn't leverage Spark's parallelization well
  - Still lots of redundant searching
  - Not recommended

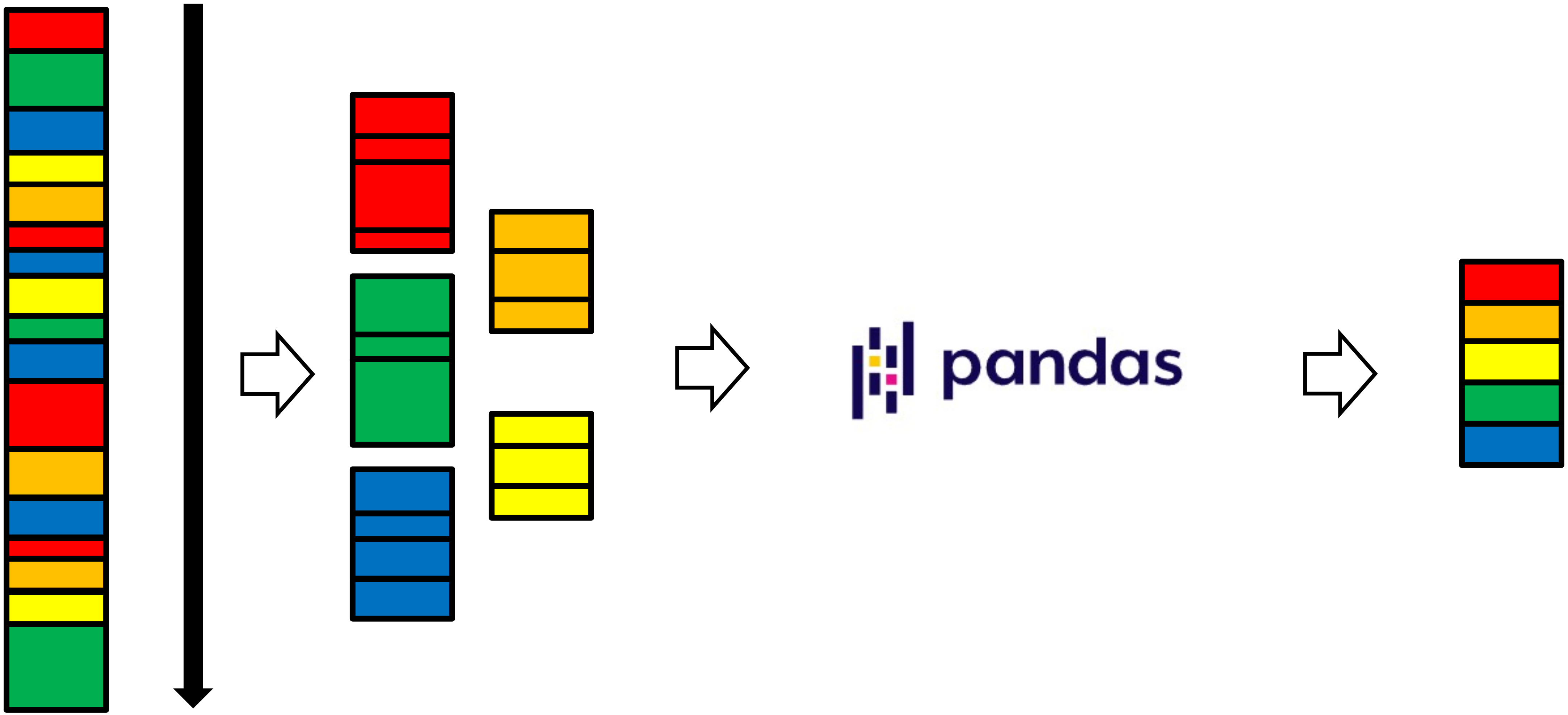


Enter PandasUDFs

PandasUDFs

- PandasUDFs let you write Python/Pandas User Defined Functions to do whatever you need.
- Significantly better performance than regular Python UDFs.
- Options:
  - Group by key, DataFrame input → DataFrame output (Grouped Map)
  - Group by key, 2 DataFrame inputs -> DataFrame output (Co-Grouped Map)
  - Group by key, Series input → value (Grouped Agg)
  - Series input → Series output (Scalar/Scalar Iter)

PandasUDFs Grouped Map



How to do a Grouped Map

```python
import joblib
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import average_precision_score

sdf = spark.sql(...)
LABEL = labelval
FEATURES = [list_of_features]
GROUPING_KEY = group_key
SAVE_FOLDER = save_folder
TEST_CATEGORY = some_value

pdf = sdf.filter(GROUPING_KEY == TEST_CATEGORY).toPandas()

X = pdf[FEATURES]
y = pdf[LABEL]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

model = (RandomForestClassifier(max_depth=3, n_estimators=50).
         fit(X_train, y_train))

y_pred = model.predict_proba(X_test)[:, 1]
pr_auc = average_precision_score(y_score=y_pred, y_true=y_test)

save_file = '/dbfs' + SAVE_FOLDER + 'key' + TEST_CATEGORY + '.joblib'
joblib.dump(model, save_file)

print(pr_auc)
```

How to do a Grouped Map

```python
import joblib
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import average_precision_score

sdf = spark.sql(...)
LABEL = labelval
FEATURES = [list_of_features]
GROUPING_KEY = group_key
SAVE_FOLDER = save_folder
TEST_CATEGORY = some_value

pdf = sdf.filter(GROUPING_KEY == TEST_CATEGORY).toPandas()

def fit_model(pdf):
    X = pdf[FEATURES]
    y = pdf[LABEL]
    category = pdf[GROUPING_KEY].iloc[0]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

    model = (RandomForestClassifier(max_depth=3, n_estimators=50).
            fit(X_train, y_train))

    y_pred = model.predict_proba(X_test)[:, 1]
    pr_auc = average_precision_score(y_score=y_pred, y_true=y_test)

    save_file = '/dbfs' + SAVE_FOLDER + 'key' + category + '.joblib'
    joblib.dump(model, save_file)

    return pr_auc
pr_auc = fit_model(pdf)
```

How to do a Grouped Map

```python
import joblib
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import average_precision_score
from pyspark.sql.functions import pandas_udf, PandasUDFType
import pandas as pd

sdf = spark.sql(...)
LABEL = labelval
FEATURES = [list_of_features]
GROUPING_KEY = group_key
SAVE_FOLDER = save_folder

@pandas_udf("category string, pr_auc double", PandasUDFType.GROUPED_MAP)
def fit_model(pdf):
    X = pdf[FEATURES]
    y = pdf[LABEL]
    category = pdf[GROUPING_KEY].iloc[0]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

    model = (RandomForestClassifier(max_depth=3, n_estimators=50).
             fit(X_train, y_train))

    y_pred = model.predict_proba(X_test)[:, 1]
    pr_auc = average_precision_score(y_score=y_pred, y_true=y_test)

    save_file = '/dbfs' + SAVE_FOLDER + 'key' + category + '.joblib'
    joblib.dump(model, save_file)

    return pd.DataFrame({'category': [category], 'pr_auc': [pr_auc]})

results = sdf.groupby(GROUPING_KEY).apply(fit_model)
results.collect()
```

What just happened?

- I took my DS code, made a function, added a decorator.
- Spark handles all the parallelization for me.
- I’m just returning a model metadata dataframe.
- Side effects (saving the model) are perfectly fine!
- No problem loading variables from script.
- My >10 hour workflow ran in 30 minutes!

Important notes

- Can define schemas with string or using elements from `pyspark.sql.types` and making a `StructType`.
- Diagnosing error messages is difficult, I recommend a try/except block to catch errors.
- Can have multiple grouping keys.
- Transformation, not an action.

```
import pyspark.sql.types as T
import pyspark.sql.functions as F

fit_schema = T.StructType([
    T.StructField("category", T.StringType(), True),
    T.StructField("pr_auc", T.DoubleType(), True),
    T.StructField("err_msg", T.StringType(), True)
])


@F.pandas_udf(fit_schema, F.PandasUDFType.GROUPED_MAP)
def fit_model(pdf):
    category = pdf[GROUPING_KEY].iloc[0]
    try:
        # Regular ML code
        return pd.DataFrame(
            {"category": [category], "pr_auc": [pr_auc], "err_msg": [None]})
    except Exception as e:
        return pd.DataFrame(
            {"category": [category], "pr_auc": [None], "err_msg": [str(e)]})
```

Scoring data

- Almost the same process.
- Loads a model.
- Returns a DataFrame with as many rows and the input.

```python
GROUPING_KEY = 'id_cat'

@pandas_udf(f"household long, {GROUPING_KEY} long, propensity float",
            PandasUDFType.GROUPED_MAP)
def predict_propensities(pdf):
    X = pdf[FEATURES]
    households = pdf['household']
    category = pdf[GROUPING_KEY].iloc[0]
    file_name = '/dbfs' + SAVE_FOLDER + 'key' + category + '.joblib'
    model = joblib.load(file_name)

    propensities = model.predict_proba(X)[:, 1]
    return pd.DataFrame({
        'household': households,
        GROUPING_KEY: [category] * len(households),
        'propensity': propensities})
```

Wrapping functions

Once the decorator is applied to the function, you can no longer run it on regular Pandas dataframes for testing.

Having a wrapping function makes this easier.

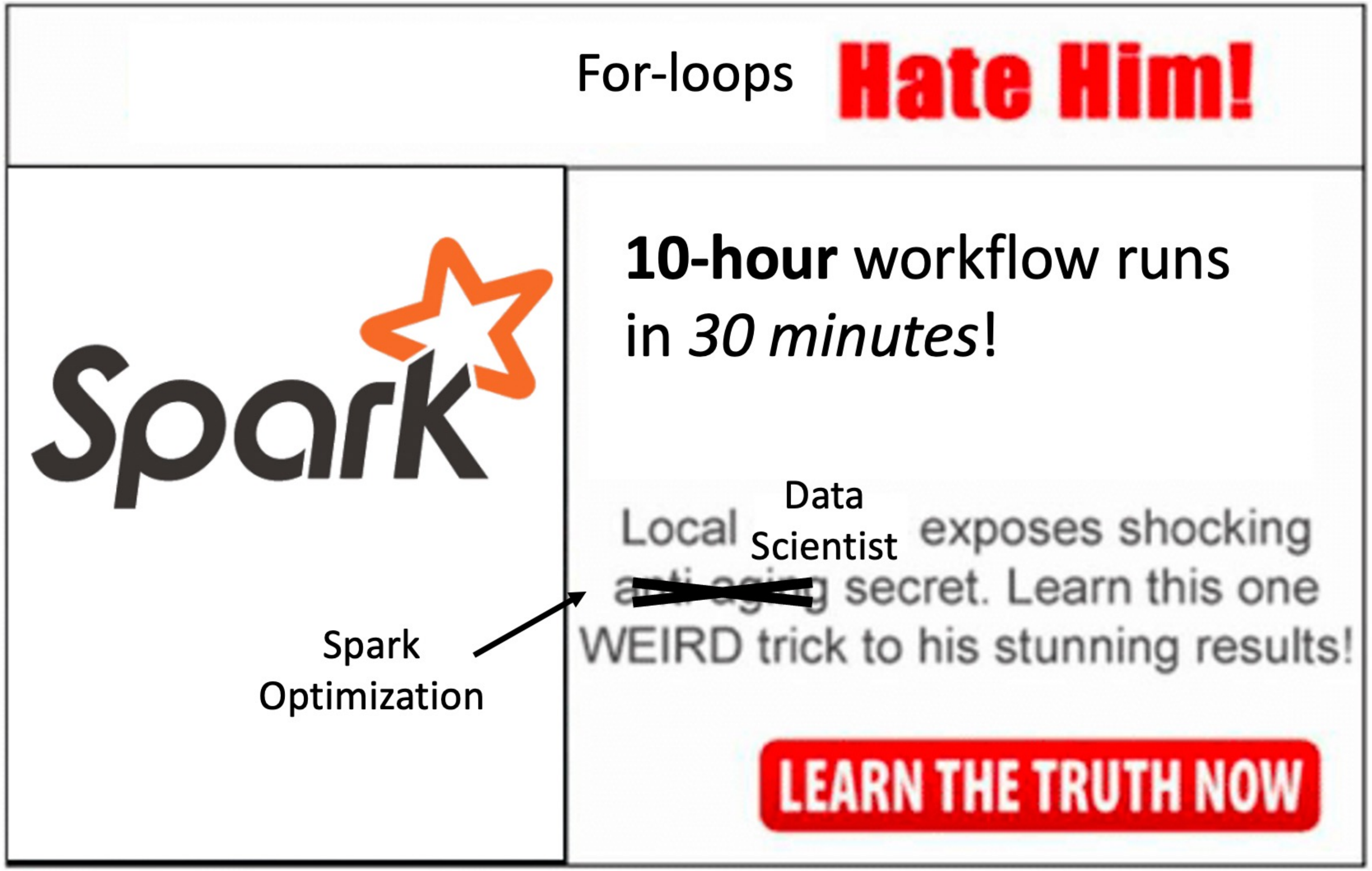
```
@pandas_udf(return_schema, PandasUDFType.GROUPED_MAP)
def fit_model_wrapper(df):
    return fit_model(df)


def fit_model(df):
    # Do ML process
    ...

output = sdf.groupby(GROUPING_KEYS).apply(fit_model_wrapper)
output.collect()
```

My one weird trick

- When it's the right tool, it does its job very well!
- I've trained ensembles from 100s to 100,000s of models.
- Used with classifiers, clusters, graphs, factor analyses, and more!



Other types of PandasUDFs

Grouped aggregations (Group Agg)

```
import numpy as np
from itertools import combinations
from pyspark.sql.functions import pandas_udf, PandasUDFType


@pandas_udf("double", PandasUDFType.GROUPED_AGG)
def HLS_estimator(series):
    # Hodges-Lehmann-Sen estimator
    sums = [sum(y) for y in combinations(series, 2)]
    return np.median(sums)/2

sdf.groupby(GROUPING_KEY).agg(HLS_estimator(df[DATA_COLUMN])).collect()
```

- Pandas Series to scalar value
- Custom aggregating function, use with `agg()` or windows.

Scalar (Scalar/Scalar Iter)

- Series → Series
- Combines well with `@np.vectorize`
- Can also use SCALAR_ITER and write generator functions.
- Only returns one value. If you need more, pack the values into a map or JSON string.

```python
import numpy as np
from numpy import linalg as lg
import pandas as pd
import pyspark.sql.functions as F


@np.vectorize
def cosine_sim(vec_a, vec_b):
    return vec_a @ vec_b / (la.norm(vec_a) * la.norm(vec_b))


@F.pandas_udf("double", F.PandasUDFType.SCALAR)
def cosine_sim_wrapper(vec_a, vec_b):
    index = vec_a.index
    return pd.Series(cosine_sim(vec_a, vec_b), index=index)


sdf = sdf.withColumn(
    cosine_sim_wrapper(F.col("vec_a"), F.col("vec_b")).
    alias("cosine_sim")
)
```

Co-Grouped Map

- 2 DataFrames input, one DataFrame output.
- Groups on same key.
- Grouped Map where you need two DataFrames at once.
- Spark 3.0+.

```
def dual_udf_function(
    pdf_1: pd.DataFrame,
    pdf_2: pd.DataFrame) -> pd.DataFrame:
    ...

final_sdf = (
    sdf_1.
    groupby(GROUPING_KEYS).
    cogroup(sdf2.groupby(GROUPING_KEYS)).
    applyInPandas(dual_udf_function, SCHEMA))
```

New syntax for Spark 3.0

- Can use Python type hints rather than defining PandasUDFType.
- For Grouped Map, there is `applyInPandas` (link) instead of apply, and we don't need to put the decorator on the function.
  - No wrapper needed.
- Co-grouped map.
- More details at here.

```python
# Old
@F.pandas_udf(SCHEMA, UDF_TYPE)
def udf_function(pdf: pd.DataFrame) -> pd.DataFrame:
    ...

sdf.groupby(GROUPING_KEYS).apply(udf_function)

# New
def udf_function(pdf: pd.DataFrame) -> pd.DataFrame:
    ...

sdf.groupby(GROUPING_KEYS).applyInPandas(
    udf_function, schema=SCHEMA)
```

When is this the right tool?

If you can express your code with pure Spark DataFrame operations, use that.

PandasUDFs are better for:

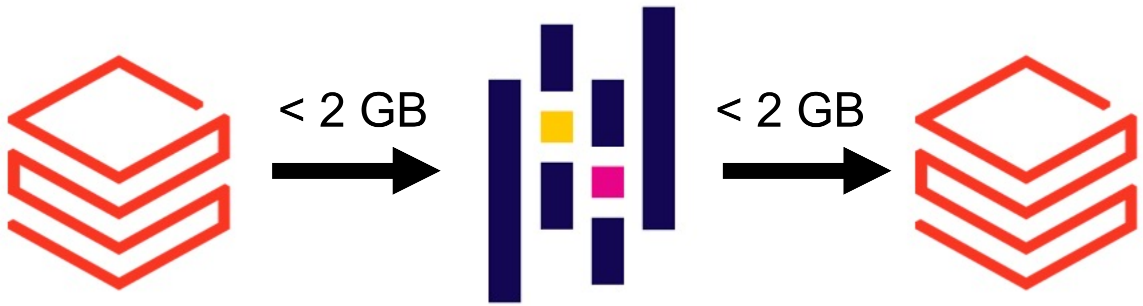
- Need some functionality you can get in Python that you can't get in PySpark.
  - Replacing regular Python UDFs.
- As a replacement for for-loops.

Image: [Pixabay]

The 2 GB limit

2 GB

- There was a 2 GB data size limit for your dataframe input and dataframe output. (Ser/De)
- Based on Arrow limits.
- Fixed in Spark 3.1+.
  - SPARK-33189
  - ARROW-10957



Workarounds

- Use lower precision data.
- Use symmetries.
  - Rare, but extremely powerful.
- Offload data to variables.

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}$$

Workarounds - subsampling

- Look at the learning curves for models to see how much model quality declines with smaller samples.
- Can be an occasion to do class balancing.
- For fitting, not scoring.

```python
import pyspark.sql.functions as F
from pyspark.sql import Window

MAX_NUMBER_OF_ROWS = ...

window = Window.partitionBy(sdf[GROUPING_KEYS]).orderBy(F.rand())
new_sdf = (sdf.
           select('*', F.rank().over(window).alias('rank')).
           filter(F.col('rank') <= MAX_NUMBER_OF_ROWS).
           drop('rank'))
```

Salting

- Simply add an extra, dummy column to your group-by key.
- Use for scoring, not fitting.
- Can also be used to partition data into more equal sized groups.

```
GROUPING_KEYS = [...]
GROUPING_KEYS.append('DUMMY_KEY')
DUMMY_KEY_CARDINALITY = 5

sdf = sdf.withColumn(
    'DUMMY_KEY',
    (DUMMY_KEY_CARDINALITY*F.rand(seed=12345)).cast("int"))

output = sdf.groupby(GROUPING_KEYS).apply(score_model)
output.collect()
```

Other frameworks

Koalas apply

- The syntax in Koalas matches what you would expect from Pandas.
- Usual Ser/De concerns.
- (Optional) Use Python type hinting to demonstrate what the schema should look like.

```python
import pandas as pd
import koalas as ks


def udf_function(pdf: pd.DataFrame) -> ks.DataFrame[SCHEMA]:
    ...


kdf.groupby(GROUPING_KEYS).apply(udf_function)
```

[Documentation link]

How about for R?

- Depends if you use SparkR or SparklyR
- If you use SparkR, `gapply` is your Grouped Map and `dapply` is your Scalar.
- If you use SparklyR, `spark_apply` does both.

https://databricks.com/blog/2018/08/15/100x-faster-bridge-between-spark-and-r-with-user-defined-functions-on-databricks.html

https://spark.rstudio.com/reference/spark_apply/

SparkR: gapply/dapply/etc

- Note the libraries initialized inside the `gapply` call.
- Can either give schema string or use `structField/structType`.
- If you have strings in your R data.frame, you need to set `stringsAsFactors=FALSE` in the data.frame construction.
- Usual Ser/De concerns.
- There is also `gapplyCollect` and `dapplyCollect`, which just is `gapply` or `dapply` with an added `collect` call.

```r
sdf = # SparkR DataFrame

fit_data <- gapply(
    sdf,
    c(GROUPING_KEY),
    function(key, x) {
        suppressMessages(library(library_call))
        df <- fit_model(key, x)
    },
    schema
)
```

SparklyR: spark_apply

- Do not need to specify schema.
- group_by parameter lets you switch between scalar and grouped-map type operations.
- Usual Ser/De concerns.
- Will be a bit slower than `gapply/dapply` – runs a final step at the end.

```r
sdf = # SparklyR DataFrame

fit_data <- spark_apply(
    sdf,
    fit_model,
    group_by = c(GROUPING_KEY)
)
```

In summary

- PandasUDFs are a powerful, situational tool.
- Let us leverage additional capabilities with Python (or R).
- Look for places where you have for-loops or are using Python UDFs.

Feedback

Your feedback is important to us.

Don’t forget to rate and review the sessions.

DATA+AI SUMMIT 2021
FORMERLY SPARK+AI SUMMIT