SPARK+AI SUMMIT 2018

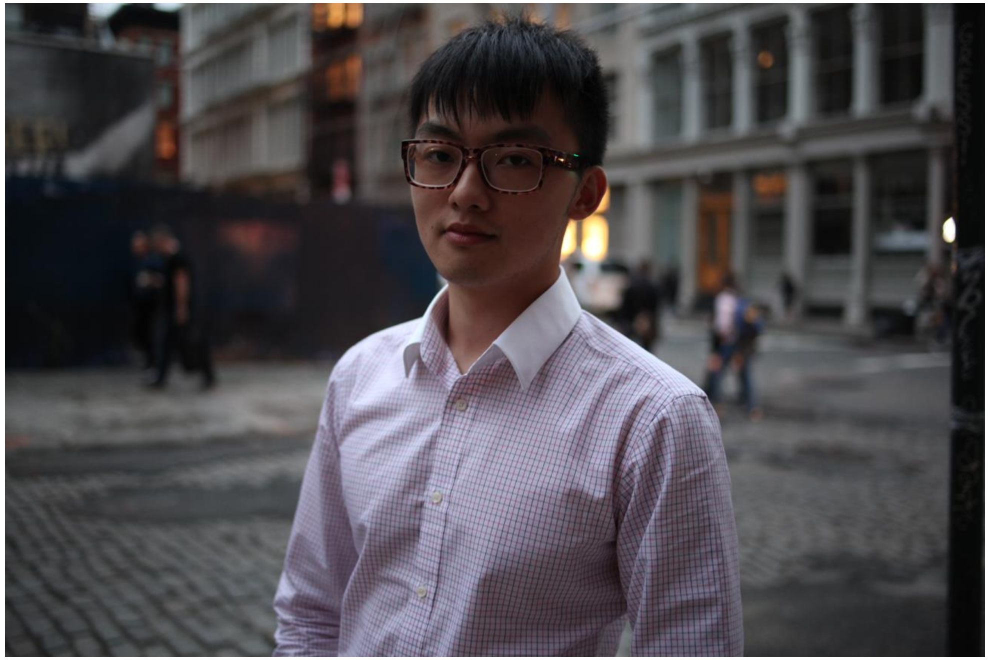
Pandas UDF

Scalable Analysis with Python and PySpark

Li Jin, Two Sigma Investments

About Me

- Li Jin (icexelloss)
- Software Engineer @ Two Sigma Investments
- Analytics Tools Smith
- Apache Arrow Committer
- Other Open Source Projects:
  - Flint: A Time Series Library on Spark



Important Legal Information

- The information presented here is offered for informational purposes only and should not be used for any other purpose (including, without limitation, the making of investment decisions). Examples provided herein are for illustrative purposes only and are not necessarily based on actual data. Nothing herein constitutes: an offer to sell or the solicitation of any offer to buy any security or other interest; tax advice; or investment advice. This presentation shall remain the property of Two Sigma Investments, LP (“Two Sigma”) and Two Sigma reserves the right to require the return of this presentation at any time.

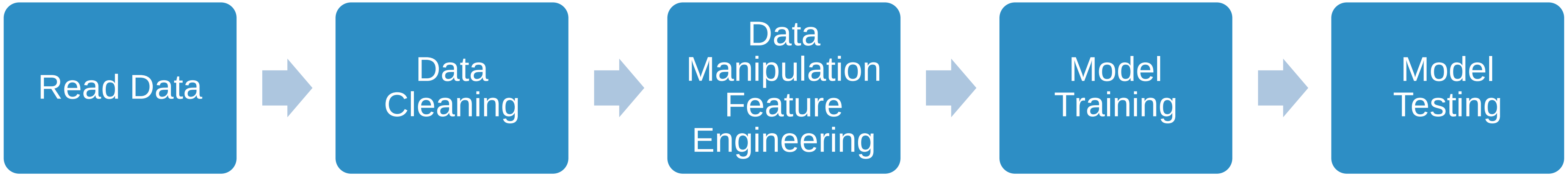
- Some of the images, logos or other material used herein may be protected by copyright and/or trademark. If so, such copyrights and/or trademarks are most likely owned by the entity that created the material and are used purely for identification and comment as fair use under international copyright and/or trademark laws. Use of such image, copyright or trademark does not imply any association with such organization (or endorsement of such organization) by Two Sigma, nor vice versa.

Outline

- Overview: Data Science in Python and Spark
- Pandas UDF in Spark 2.3
- Ongoing work

Overview: Data Science in Python and Spark

Predictive Modeling



Predictive Modeling (Python)

Read Data	Data Cleaning	Data Manipulation Feature Engineering	Model Training	Model Testing
pandas	pandas numpy	pandas numpy scipy	sklearn	sklearn

Predictive Modeling (Spark)

Read Data → Data Cleaning → Data Manipulation Feature Engineering → Model Training → Model Testing

Read Data	Data Cleaning	Data Manipulation Feature Engineering	Model Training	Model Testing
Spark SQL	Spark SQL	Spark SQL Spark ML	Spark ML	Spark ML

The Problem…Feature Gap

- Many functionality in Python is not **available** or **easy** in Spark

Stack Overflow Answer: Forward Fill (Python)

You could use the `fillna` method on the DataFrame and specify the method as `ffill` (forward fill):

```
>>> df = pd.DataFrame([[1, 2, 3], [4, None, None], [None, None, 9]])
>>> df.fillna(method='ffill')
   0  1  2
0  1  2  3
1  4  2  3
2  4  2  9
```

This method...

> propagate[s] last valid observation forward to next valid

To go the opposite way, there's also a `bfill` method.

This method doesn't modify the DataFrame inplace - you'll need to rebind the returned DataFrame to a variable or else specify `inplace=True`:

```
df.fillna(method='ffill', inplace=True)
```

Stack Overflow Answer: Forward Fill (Spark)

Edit (partitioned / time series per group data):

The devil is in the detail. If your data is partitioned after all then a whole problem can be solved using `groupBy`. Lets assume you simply partition by column "v" of type `T` and `Date` is an integer timestamp:

```
def fill(iter: List[Row]): List[Row] = {
  // Just go row by row and fill with last non-empty value
  ???
}

val groupedAndSorted = df.rdd
  .groupBy(_.getAs[T]("k"))
  .mapValues(_.toList.sortBy(_.getAs[Int]("Date")))

val rows: RDD[Row] = groupedAndSorted.mapValues(fill).values.flatMap(identity)

val dfFilled = sqlContext.createDataFrame(rows, df.schema)
```

This way you can fill all columns at the same time.

> Can this be done with DataFrames instead of converting back and forth to RDD?

It depends, although it is unlikely to be efficient. If maximum gap is relatively small you can do something like this:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.{WindowSpec, Window}
import org.apache.spark.sql.Column

val maxGap: Int = ???  // Maximum gap between observations
val columnsToFill: List[String] = ???  // List of columns to fill
val suffix: String = "_" // To disambiguate between original and imputed

// Take lag 1 to maxGap and coalesce
def makeCoalesce(w: WindowSpec)(magGap: Int)(suffix: String)(c: String) = {
  // Generate lag values between 1 and maxGap
  val lags = (1 to maxGap).map(lag(col(c), _)over(w))
```

```
// Finally select
val dfImputed = df.select($"*" :: lags: _*)
```

It can be easily adjusted to use different maximum gap per column.

A simpler way to achieve a similar result in the latest Spark version is to use `last` with `ignoreNulls`:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

val w = Window.partitionBy($"k").orderBy($"Date")
  .rowsBetween(Window.unboundedPreceding, -1)

df.withColumn("value", coalesce($"value", last($"value", true).over(w)))
```

While it is possible to drop `partitionBy` clause and apply this method globally, it would prohibitively expensive with large datasets.

Stack Overflow Answer: Forward Fill (Spark)

Edit (partitioned / time series per group data):

The devil is in the detail. If your data is partitioned after all then a whole problem can be solved using `groupBy`. Lets assume you simply partition by column "v" of type `T` and `Date` is an integer timestamp:

```
def fill(iter: List[Row]): List[Row] = {
  // Just go row by row and fill with last non-empty value
  ???
}

val groupedAndSorted = df.rdd
  .groupBy(_.getAs[T]("k"))
  .mapValues(_.toList.sortBy(_.getAs[Int]("Date")))

val rows: RDD[Row] = groupedAndSorted.mapValues(fill).values.flatMap(identity)

val dfFilled = sqlContext.createDataFrame(rows, df.schema)
```

This way you can fill all columns at the same time.

> Can this be done with DataFrames instead of converting back and forth to RDD?

It depends, although it is unlikely to be efficient. If maximum gap is relatively small you can do something like this:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.{WindowSpec, Window}
import org.apache.spark.sql.Column

val maxGap: Int = ???  // Maximum gap between observations
val columnsToFill: List[String] = ???  // List of columns to fill
val suffix: String = "_" // To disambiguate between original and imputed

// Take lag 1 to maxGap and coalesce
def makeCoalesce(w: WindowSpec)(magGap: Int)(suffix: String)(c: String) = {
  // Generate lag values between 1 and maxGap
  val lags = (1 to maxGap).map(lag(col(c), _)over(w))
```

```
// Finally select
val dfImputed = df.select($"*" :: lags: _*)
```

It can be easily adjusted to use different maximum gap per column.

A simpler way to achieve a similar result in the latest Spark version is to use `last` with `ignoreNulls`:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window

val w = Window.partitionBy($"k").orderBy($"Date")
  .rowsBetween(Window.unboundedPreceding, -1)

df.withColumn("value", coalesce($"value", last($"value", true).over(w)))
```

While it is possible to drop `partitionBy` clause and apply this method globally, it would prohibitively expensive with large datasets.

Feature Gap: Forward Fill

- Spark SQL:
  - Previous/Next observation
- Python:
  - Previous/Next observation
  - Interpolation
    - Linear
    - Quadratic
    - …

Feature Gap between Spark and Python

- Data Cleaning and Manipulation
  - Fill missing values (pandas.DataFrame.fillna)
  - Rank features (scipy.stats.percentileofscore)
  - Exponential moving average (pandas.DataFrame.ewm)
  - Power transformations (scipy.stats.boxcox)
  - …
- Modeling Training
  - …

Spark and Python

Spark
Scalable

?

Python
Functionality

Pandas UDF in Spark 2.3

Strength of Spark and Python

- How (Spark SQL)
  - For each row
  - For each group
  - Over rolling window
  - Over entire data
  - …
- What (Python)
  - Filling missing value
  - Rank features
  - …

Combine What and How: PySpark UDF

- Interface for extending Spark with native Python libraries
- UDF is executed in a separate Python process
- Data is transferred between Python and Java

Existing UDF

- Python function on each Row
- Data serialized using Pickle
- Data as Python objects (Python integer, Python lists, …)

Existing UDF (Functionality)

- How (Spark SQL)
  - For each row
  - For each group
  - Over rolling window
  - Over entire data
  - …
- What (Python)
  - Filling missing value
  - Rank features
  - …

Most relational functionality is taken away

Existing UDF (Usability)

v – v.mean() / v.std()

groupby year month

```
group_columns = ['year', 'month']
non_group_columns = [col for col in df.columns if col not in group_columns]
s = StructType([f for f in df.schema.fields if f.name in non_group_columns])
cols = list([F.col(name) for name in non_group_columns])

df_norm = df.withColumn('values', F.struct(*cols))
df_norm = (df_norm.groupBy('year', 'month')
                  .agg(F.collect_list(df_norm.values).alias('values')))

s2 = StructType(s.fields + [StructField('v3', DoubleType())])
@udf(ArrayType(s2))
def normalize(values):
    v1 = pd.Series([r.v1 for r in values])
    v1_norm = (v1 - v1.mean()) / v1.std()
    return [values[i] + (float(v1_norm[i]),) for i in range(0, len(values))]

df_norm = (df_norm.withColumn('new_values', normalize(df_norm.values))
                  .drop('values')
                  .withColumn('new_values', F.explode(F.col('new_values'))))

for col in [f.name for f in s2.fields]:
    df_norm = df_norm.withColumn(col, F.col('new_values.{0}'.format(col)))

df_norm = df_norm.drop('new_values')
```

Existing UDF (Usability)

80% of the code is boilerplate

```
group_columns = ['year', 'month']
non_group_columns = [col for col in df.columns if col not in group_columns]
s = StructType([f for f in df.schema.fields if f.name in non_group_columns])
cols = list([F.col(name) for name in non_group_columns])

df_norm = df.withColumn('values', F.struct(*cols))
df_norm = (df_norm.groupBy('year', 'month')
              .agg(F.collect_list(df_norm.values).alias('values')))

s2 = StructType(s.fields + [StructField('v3', DoubleType())])
@udf(ArrayType(s2))
def normalize(values):
    v1 = pd.Series([r.v1 for r in values])
    v1_norm = (v1 - v1.mean()) / v1.std()
    return [values[i] + (float(v1_norm[i]),) for i in range(0, len(values))]

df_norm = (df_norm.withColumn('new_values', normalize(df_norm.values))
              .drop('values')
              .withColumn('new_values', F.explode(F.col('new_values'))))

for col in [f.name for f in s2.fields]:
    df_norm = df_norm.withColumn(col, F.col('new_values.{0}'.format(col)))

df_norm = df_norm.drop('new_values')
```

Existing UDF (Performance)

8 Mb/s

Profile UDF
lambda x: x + 1

```
8787091 function calls in 4.084 seconds

Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  20973    1.296    0.000    3.820    0.000 serializers.py:223(_batched)
2097152    0.800    0.000    2.004    0.000 worker.py:107(<lambda>)
2097152    0.761    0.000    1.204    0.000 worker.py:72(<lambda>)
2097152    0.443    0.000    0.443    0.000 <ipython-input-2-853f857cd265>:14(<lambda>)
2097152    0.214    0.000    0.214    0.000 {method 'append' of 'list' objects}
  20972    0.153    0.000    0.153    0.000 {built-in method _pickle.loads}
  20972    0.086    0.000    0.086    0.000 {built-in method _pickle.dumps}
  20972    0.046    0.000    0.230    0.000 serializers.py:148(_write_with_length)
  41944    0.045    0.000    0.045    0.000 {method 'write' of '_io.BufferedWriter' objects}
  20973    0.044    0.000    0.287    0.000 serializers.py:161(_read_with_length)
  41945    0.039    0.000    0.039    0.000 {method 'read' of '_io.BufferedReader' objects}
      1    0.034    0.034    4.084    4.084 serializers.py:137(dump_stream)
  20973    0.021    0.000    0.039    0.000 serializers.py:598(read_int)
  20972    0.020    0.000    0.042    0.000 serializers.py:605(write_int)
  20973    0.020    0.000    0.306    0.000 serializers.py:141(load_stream)
  20972    0.019    0.000    0.172    0.000 serializers.py:474(loads)
  20972    0.017    0.000    0.103    0.000 serializers.py:470(dumps)
  62916    0.011    0.000    0.011    0.000 {built-in method builtins.len}
  20972    0.009    0.000    0.009    0.000 {built-in method _struct.pack}
  20973    0.008    0.000    0.008    0.000 {built-in method _struct.unpack}
      1    0.000    0.000    0.000    0.000 serializers.py:246(load_stream)
      1    0.000    0.000    4.084    4.084 serializers.py:243(dump_stream)
      1    0.000    0.000    4.084    4.084 worker.py:217(process)
      1    0.000    0.000    0.000    0.000 serializers.py:249(_load_stream_without_unbatching)
      1    0.000    0.000    0.000    0.000 worker.py:121(<lambda>)
      1    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
      1    0.000    0.000    0.000    0.000 {built-in method from_iterable}
```

91.8% in
Ser/Deser

Challenge

- More expressive API
- Efficient data transfer between Java and Python (Serialization)
- Efficient data operation in Python

Pandas UDF in Spark 2.3: Scalar and Grouped Map

Existing UDF vs Pandas UDF

Existing UDF

- Function on Row
- Pickle serialization
- Data as Python objects

Pandas UDF

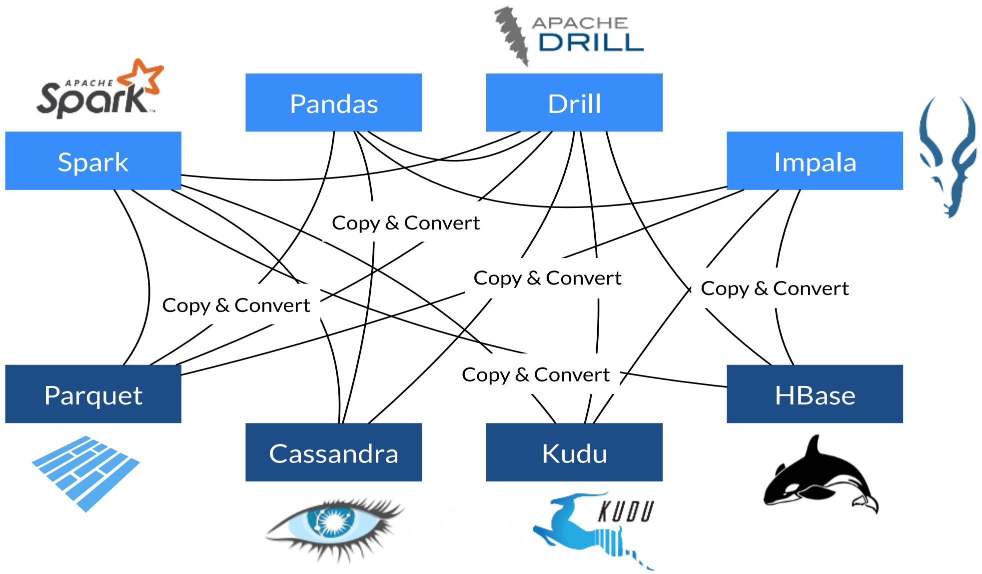
- Function on Row, Group and Window
- Arrow serialization
- Data as pd.Series (for column) and pd.DataFrame (for table)

Apache Arrow

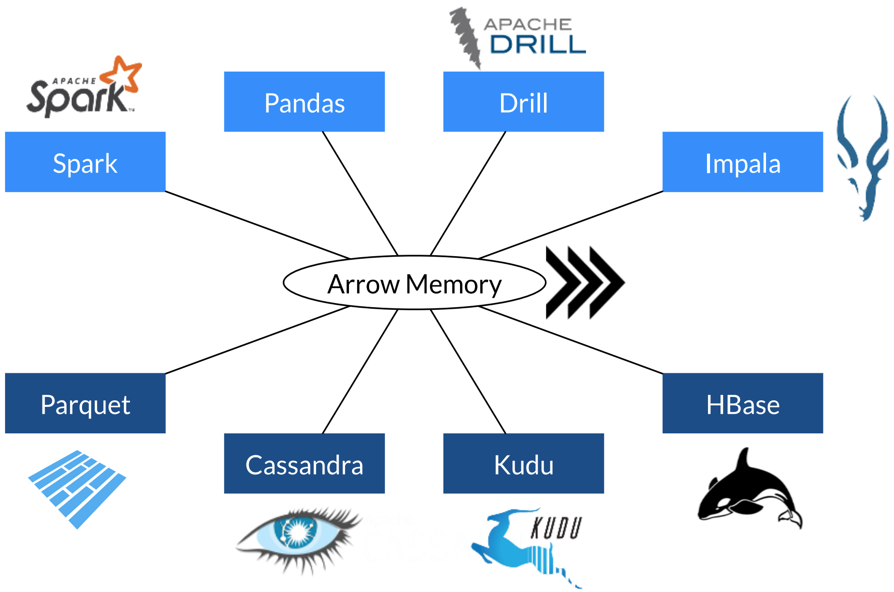
- In memory columnar format for data analysis
- Low cost to transfer between systems

Apache Arrow

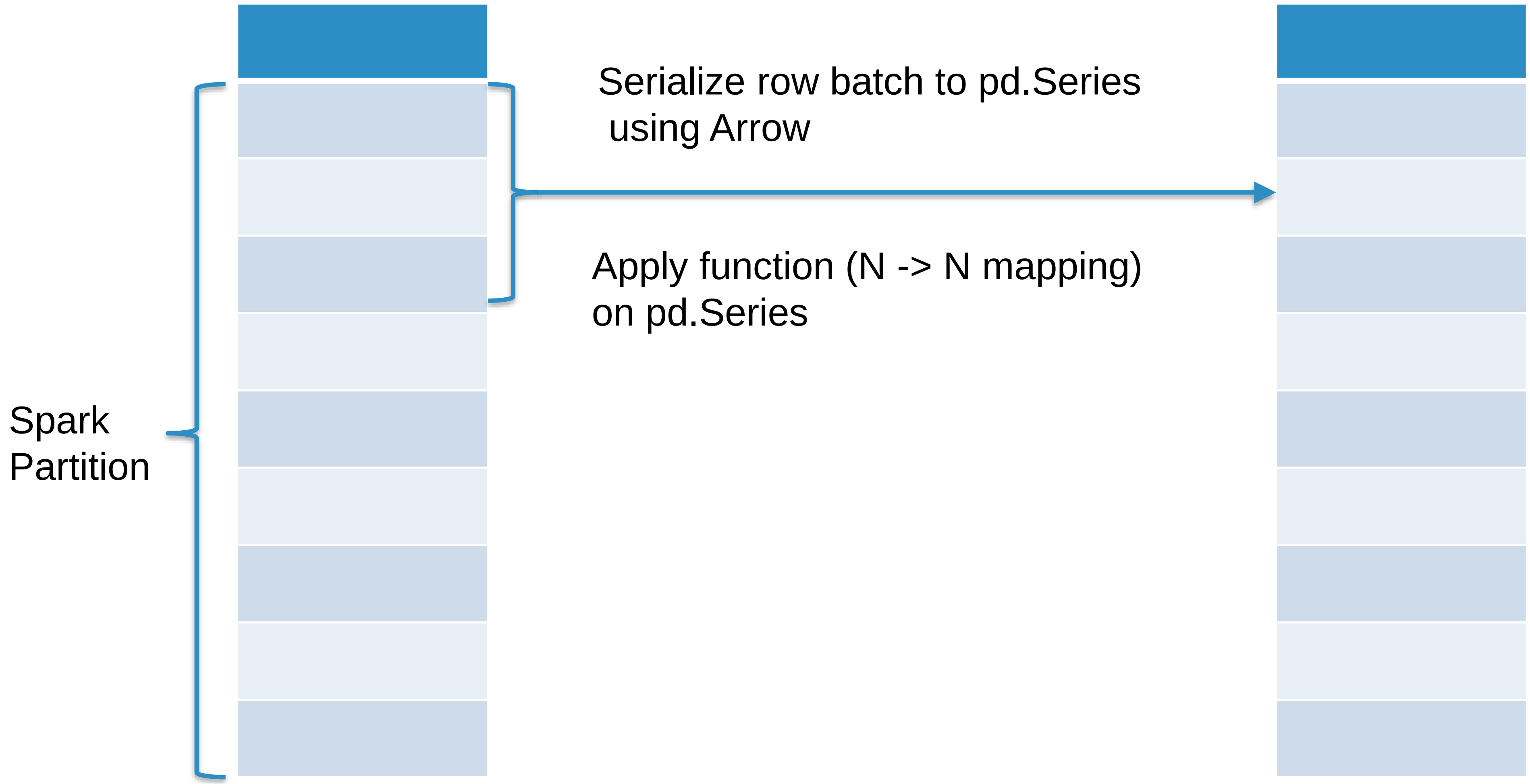
Before



With Arrow



Scalar


Serialize row batch to pd.Series
using Arrow
Apply function (N -> N mapping)
on pd.Series
Spark
Partition

Scalar Example: millisecond to timestamp

```
import pandas as pd

@pandas_udf('timestamp', PandasUDFType.SCALAR)
def millisToTimestamp(t):
    return pd.to_datetime(t, unit='ms')

df = df.withColumn('time', millisToTimestamp(df['time']))
```

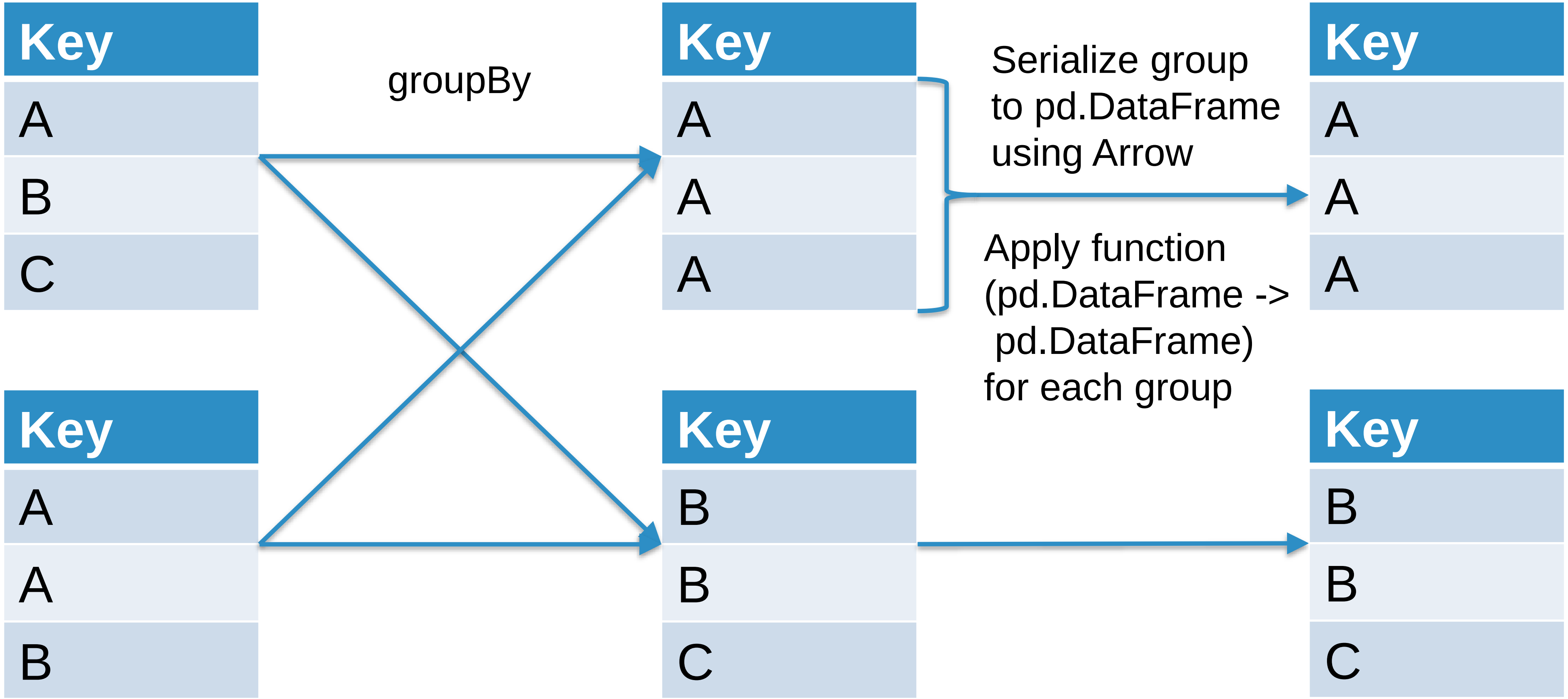
Scalar Example: cumulative density function

```python
import pandas as pd
from scipy import stats

@pandas_udf('double', PandasUDFType.SCALAR)
def cdf(v):
    return pd.Series(stats.norm.cdf(v))

df = df.withColumn('p', cdf(df.v))
```

Grouped Map

- Operations on Groups of Rows
  - Each group: N -> Any
  - Similar to flatMapGroups and “groupby apply” in Pandas

Grouped Map


Key
A
B
C
Key
A
A
B
groupBy
Key
A
A
A
Key
B
B
C
Serialize group
to pd.DataFrame
using Arrow
Apply function
(pd.DataFrame ->
pd.DataFrame)
for each group
Key
A
A
A
Key
B
B
C

Grouped Map Example: Backward Fill

```
@pandas_udf(df.schema, PandasUDFType.GROUPED_MAP)
def bfill(pdf):
    return pdf.bfill()

df = df.groupby('id').apply(bfill)
```

Grouped Map Example: Model Fitting

```
import pandas as pd
import statsmodels.api as sm
# df has four columns: id, y, x1, x2

group_column = 'id'
y_column = 'y'
x_columns = ['x1', 'x2']
const_column = 'const'
schema = 'id int, const double, ' + ", ".join("%s double" % x for x in x_columns)

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
# Input/output are both a pandas.DataFrame
def ols(pdf):
    group_key = pdf[group_column].iloc[0]
    y = pdf[y_column]
    X = pdf[x_columns]
    X = sm.add_constant(X)
    model = sm.OLS(y, X).fit()
    return pd.DataFrame([[group_key, model.params[const_column]] + [model.params[i] for i in x_columns]])

models = df.groupby(group_column).apply(ols)
```

Grouped Map Example: Model Fitting

Define constants and output schema

```
import pandas as pd
import statsmodels.api as sm
# df has four columns: id, y, x1, x2

group_column = 'id'
y_column = 'y'
x_columns = ['x1', 'x2']
const_column = 'const'
schema = 'id int, const double, ' + ", ".join("%s double" % x for x in x_columns)

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
# Input/output are both a pandas.DataFrame
def ols(pdf):
    group_key = pdf[group_column].iloc[0]
    y = pdf[y_column]
    X = pdf[x_columns]
    X = sm.add_constant(X)
    model = sm.OLS(y, X).fit()
    return pd.DataFrame([[group_key, model.params[const_column]] + [model.params[i] for i in x_columns]])

models = df.groupby(group_column).apply(ols)
```

Grouped Map Example: Model Fitting

Define model (linear regression)

```
import pandas as pd
import statsmodels.api as sm
# df has four columns: id, y, x1, x2

group_column = 'id'
y_column = 'y'
x_columns = ['x1', 'x2']
const_column = 'const'
schema = 'id int, const double, ' + ", ".join("%s double" % x for x in x_columns)

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
# Input/output are both a pandas.DataFrame
def ols(pdf):
    group_key = pdf[group_column].iloc[0]
    y = pdf[y_column]
    X = pdf[x_columns]
    X = sm.add_constant(X)
    model = sm.OLS(y, X).fit()
    return pd.DataFrame([[group_key, model.params[const_column]] + [model.params[i] for i in x_columns]])

models = df.groupby(group_column).apply(ols)
```

Improvements and limitations

Improvement (Usability)

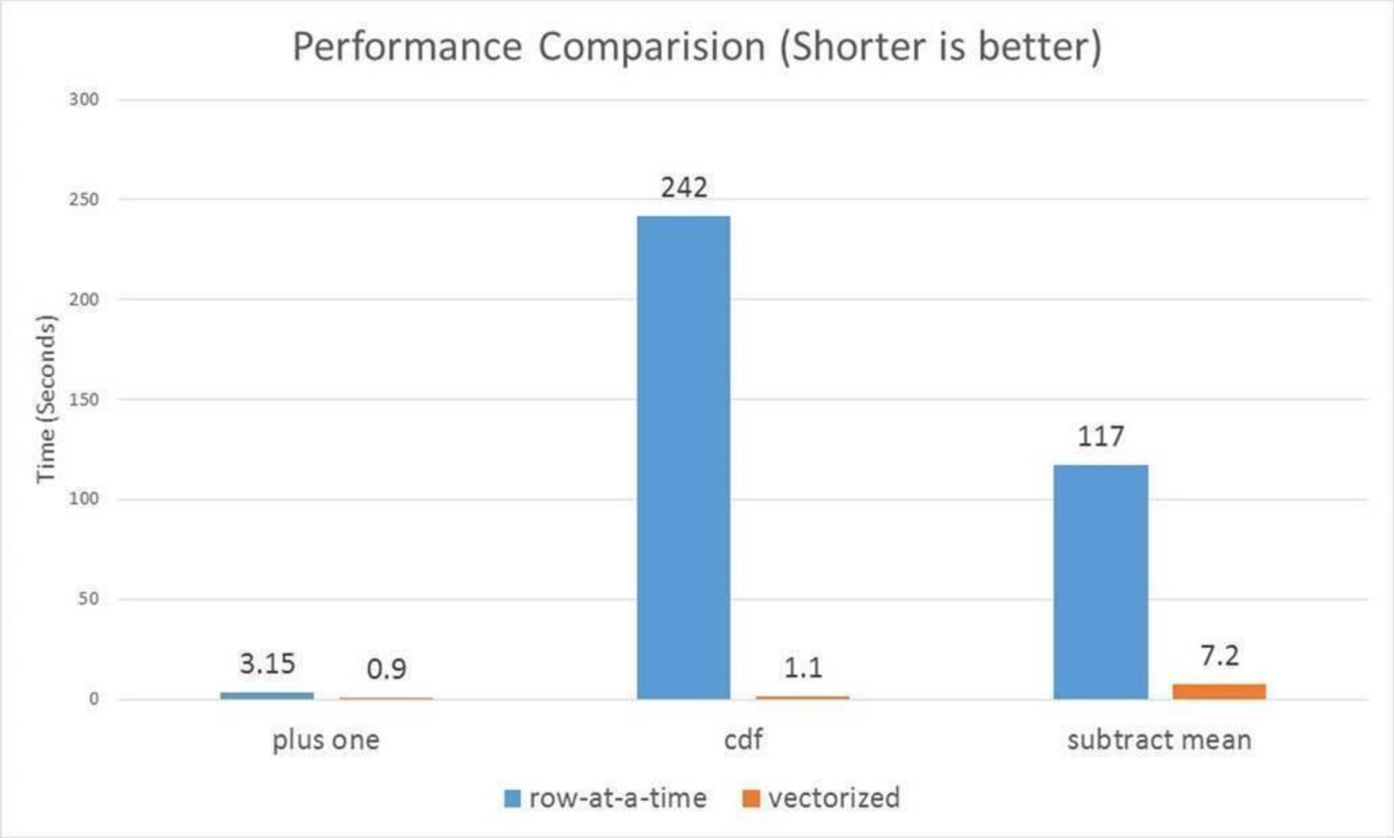
Before

```
group_columns = ['year', 'month']
non_group_columns = [col for col in df.columns if col not in group_columns]
s = StructType([f for f in df.schema.fields if f.name in non_group_columns])
cols = list([F.col(name) for name in non_group_columns])

df_norm = df.withColumn('values', F.struct(*cols))
df_norm = (df_norm.groupBy('year', 'month')
                  .agg(F.collect_list(df_norm.values).alias('values')))

s2 = StructType(s.fields + [StructField('v3', DoubleType())])
@udf(ArrayType(s2))
def normalize(values):
    v1 = pd.Series([r.v1 for r in values])
    v1_norm = (v1 - v1.mean()) / v1.std()
    return [values[i] + (float(v1_norm[i]),) for i in range(0, len(values))]

df_norm = (df_norm.withColumn('new_values', normalize(df_norm.values))
                  .drop('values')
                  .withColumn('new_values', F.explode(F.col('new_values'))))

for col in [f.name for f in s2.fields]:
    df_norm = df_norm.withColumn(col, F.col('new_values.{0}'.format(col)))

df_norm = df_norm.drop('new_values')
```

After

```
schema = StructType(df.schema.fields + [StructField('v3', DoubleType())])

@pandas_udf(schema, PandasUDFType.GROUPED_MAP)
def normalize(pdf):
    v1 = pdf.v1
    pdf['v3'] = (v1 - v1.mean()) / v1.std()
    return pdf

df_norm = df.groupby('year', 'month').apply(normalize)
```

Improvement (Performance)



https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html

Pandas UDF limitations

- Must split data
- (Grouped Map) Each group must fit entirely in memory

Ongoing Work

Pandas UDF Roadmap

- Spark-22216
- Released in Spark 2.3
  - Scalar
  - Grouped Map
- Ongoing
  - Grouped Aggregate (not yet released)
  - Window (work in progress)
  - Memory efficiency
  - Complete type support (struct type, map type)

Thank you