TWO SIGMA dremio

Improving Python and Spark Performance and Interoperability with Apache Arrow

Julien Le Dem
Principal Architect
Dremio

Li Jin
Software Engineer
Two Sigma Investments

About Us

TWO SIGMA

Li Jin

@icexelloss

- Software Engineer at Two Sigma Investments
- Building a python-based analytics platform with PySpark
- Other open source projects:
  - Flint: A Time Series Library on Spark
  - Cook: A Fair Share Scheduler on Mesos

dremio

Julien Le Dem

@J_

- Architect at @DremioHQ
- Formerly Tech Lead at Twitter on Data Platforms
- Creator of Parquet
- Apache member
- Apache PMCs: Arrow, Kudu, Incubator, Pig, Parquet

Agenda

- Current state and limitations of PySpark UDFs
- Apache Arrow overview
- Improvements realized
- Future roadmap

Current state and limitations of PySpark UDFs

Why do we need User Defined Functions?

- Some computation is more easily expressed with Python than Spark built-in functions.
- Examples:
  - weighted mean
  - weighted correlation
  - exponential moving average

What is PySpark UDF

- PySpark UDF is a user defined function executed in **Python runtime**.
- Two types:
  - Row UDF:
    - lambda x: x + 1
    - lambda date1, date2: (date1 - date2).years
  - Group UDF (subject of this presentation):
    - lambda values: np.mean(np.array(values))

Row UDF

- Operates on a row by row basis
  - Similar to `map` operator
- Example …

```
df.withColumn(
      'v2',
      udf(lambda x: x+1, DoubleType())(df.v1)
)
```

- Performance:
  - **60x** slower than build-in functions for simple case

Group UDF

- UDF that operates on more than one row
  - Similar to `groupBy` followed by `map` operator
- Example:
  - Compute weighted mean by month

Group UDF

- Not supported out of box:
  - Need boiler plate code to pack/unpack multiple rows into a nested row
- Poor performance
  - Groups are materialized and then converted to Python data structures

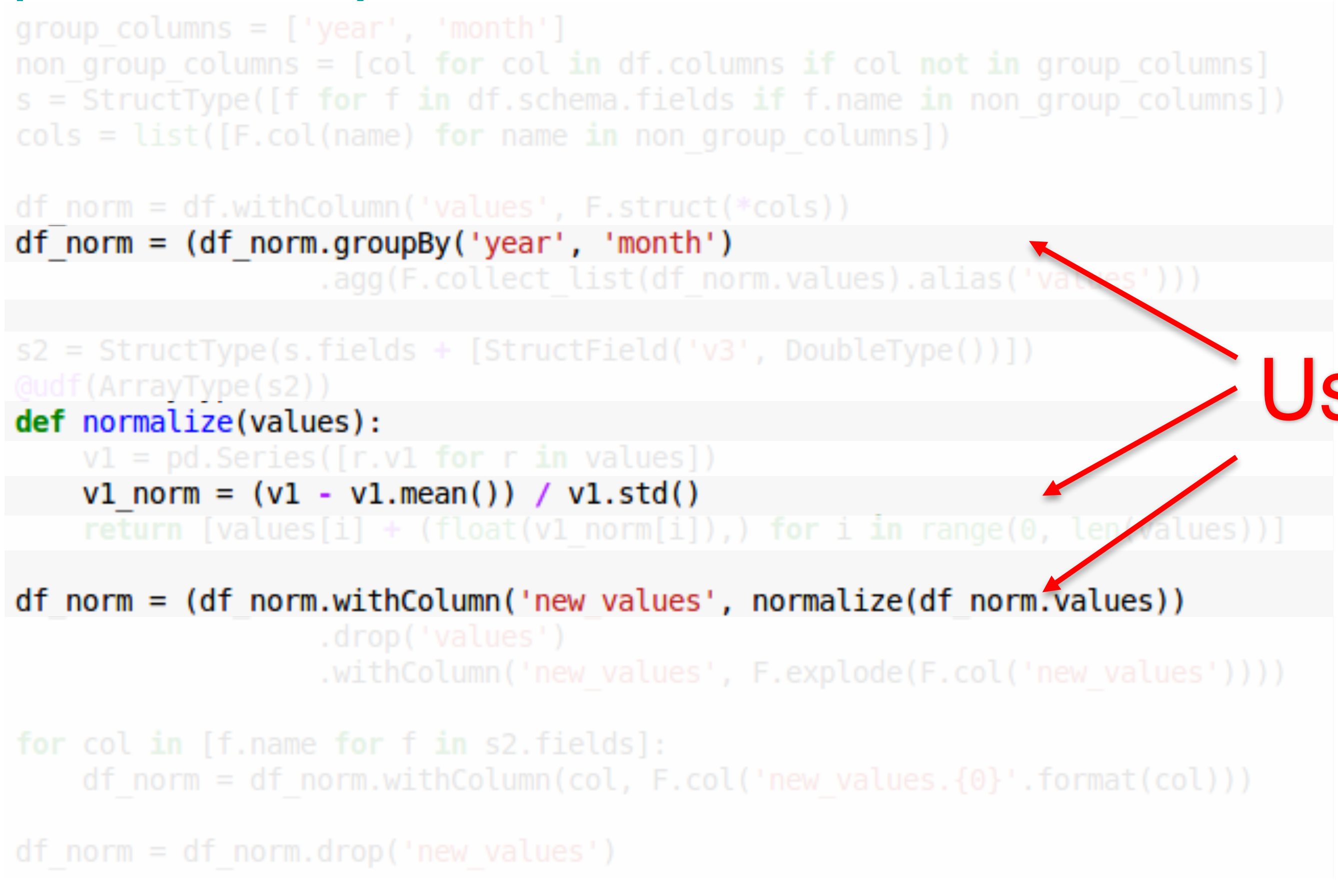
Example: Data Normalization

(values - values.mean()) / values.std()

Example: Data Normalization

```
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')
```

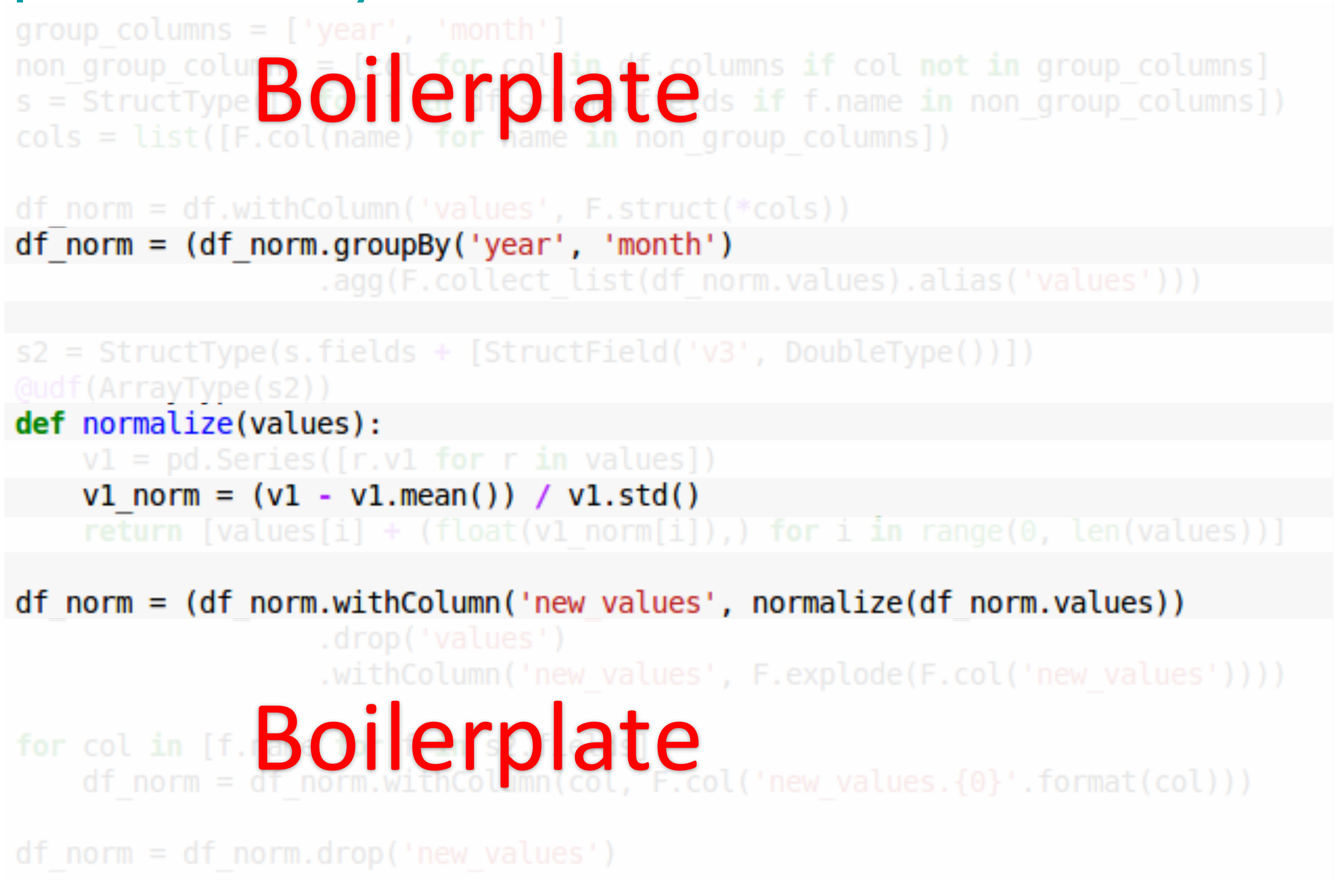
Example: Monthly Data Normalization



```python
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')
```

Useful bits

Example: Monthly Data Normalization



```
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')
```

Example: Monthly Data Normalization

- Poor performance - 16x slower than baseline

```
groupBy().agg(collect_list())
```

Problems

- Packing / unpacking nested rows
- Inefficient data movement (Serialization / Deserialization)
- Scalar computation model: object boxing and interpreter overhead

Apache Arrow

Arrow: An open source standard

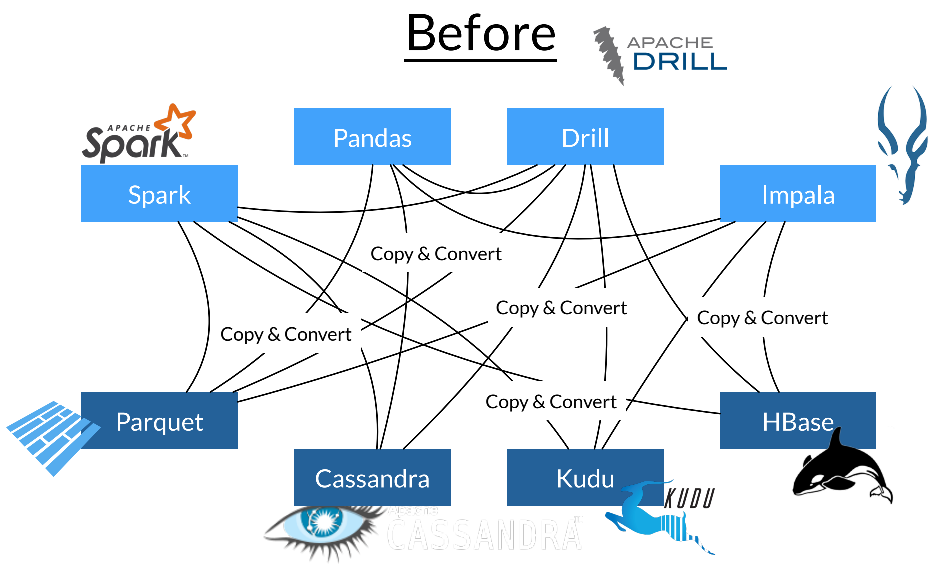
- Common need for in memory columnar
- Building on the success of Parquet.
- Top-level Apache project
- Standard from the start
  - Developers from 13+ major open source projects involved
- Benefits:
  - Share the effort
  - Create an ecosystem

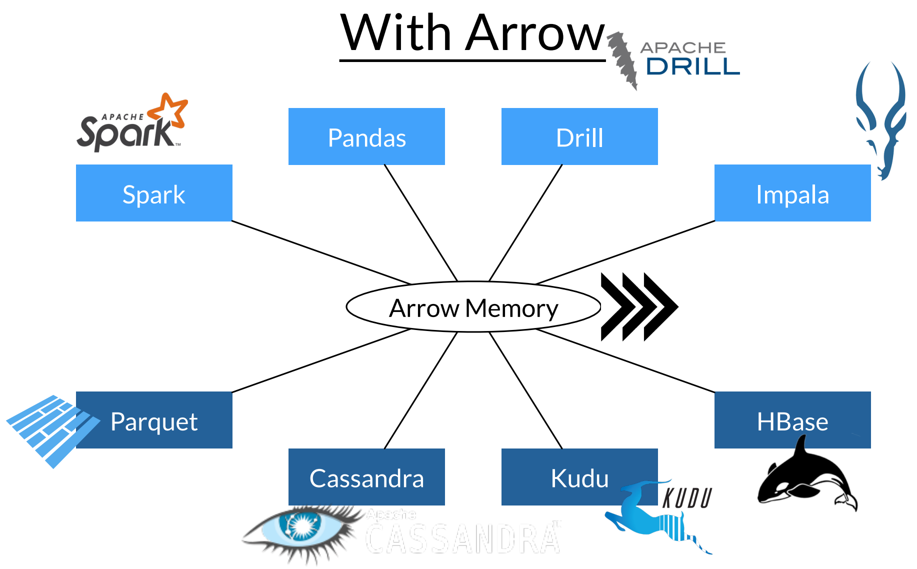
Calcite
Cassandra
Deeplearning4j
Drill
Hadoop
HBase
Ibis
Impala
Kudu
Pandas
Parquet
Phoenix
Spark
Storm
R

Arrow goals

- Well-documented and cross language compatible
- Designed to take advantage of modern CPU
- Embeddable
  - In execution engines, storage layers, etc.
- Interoperable

High Performance Sharing & Interchange



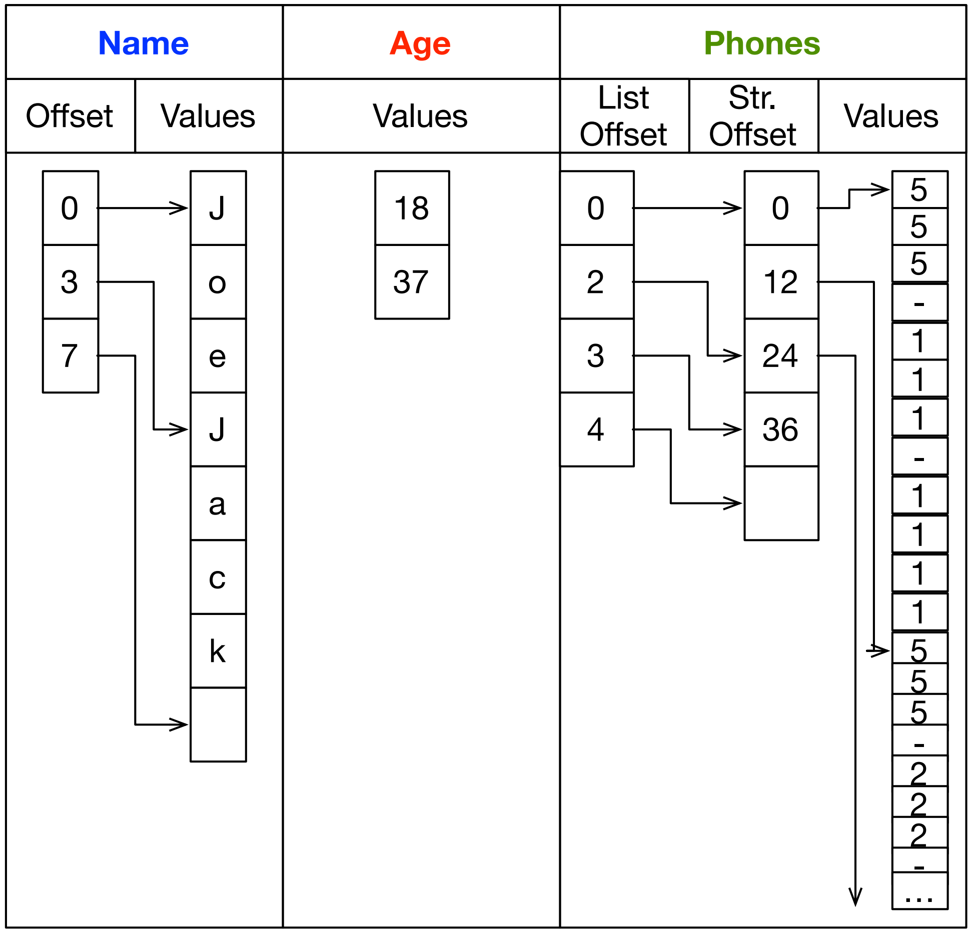


- Each system has its own internal memory format
- 70-80% CPU wasted on serialization and deserialization
- Functionality duplication and unnecessary conversions

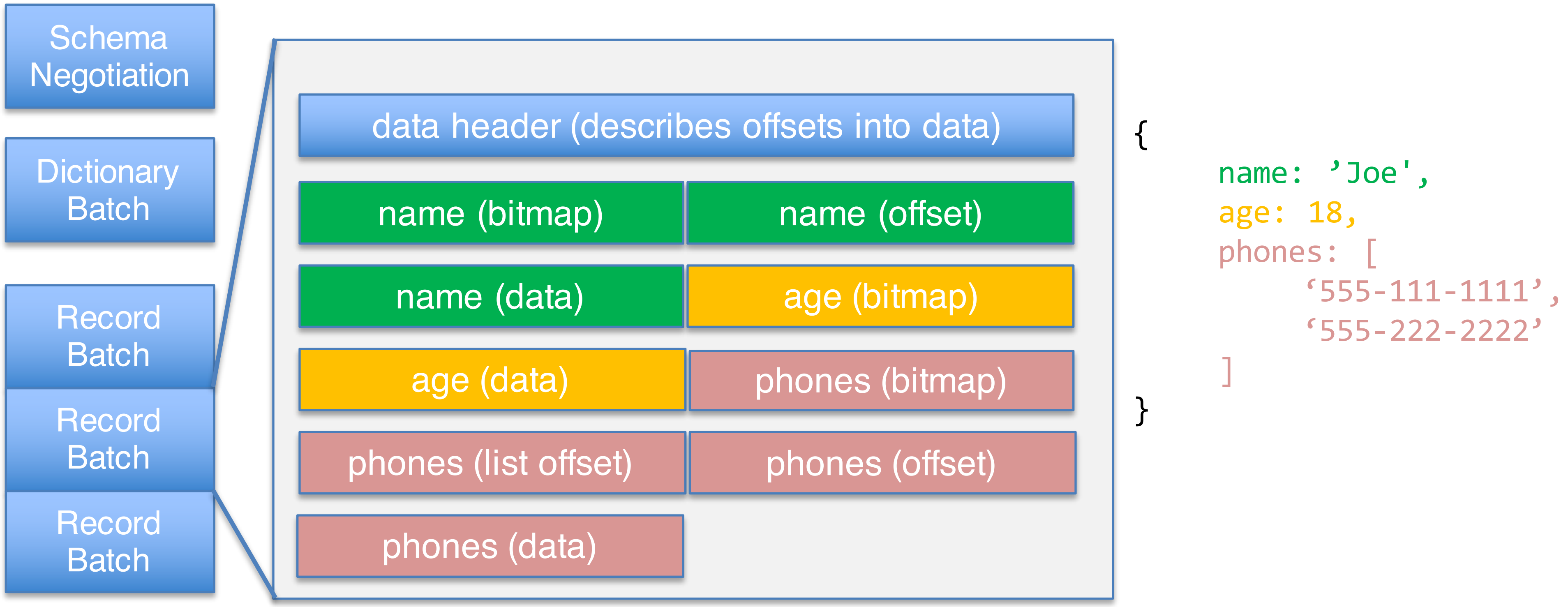
- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg: Parquet-to-Arrow reader)

Columnar data

```
persons = [{
    name: ’Joe',
    age: 18,
    phones: [
        ‘555-111-1111’,
        ‘555-222-2222’
    ]
}, {
    name: ’Jack',
    age: 37,
    phones: [ ‘555-333-3333’ ]
}]
```



Record Batch Construction



```
{
    name: ’Joe',
    age: 18,
    phones: [
        ‘555-111-1111’,
        ‘555-222-2222’
    ]
}
```

Each box (vector) is contiguous memory
The entire record batch is contiguous on wire

In memory columnar format for speed

- Maximize CPU throughput
  - Pipelining
  - SIMD
  - cache locality
- Scatter/gather I/O

Results

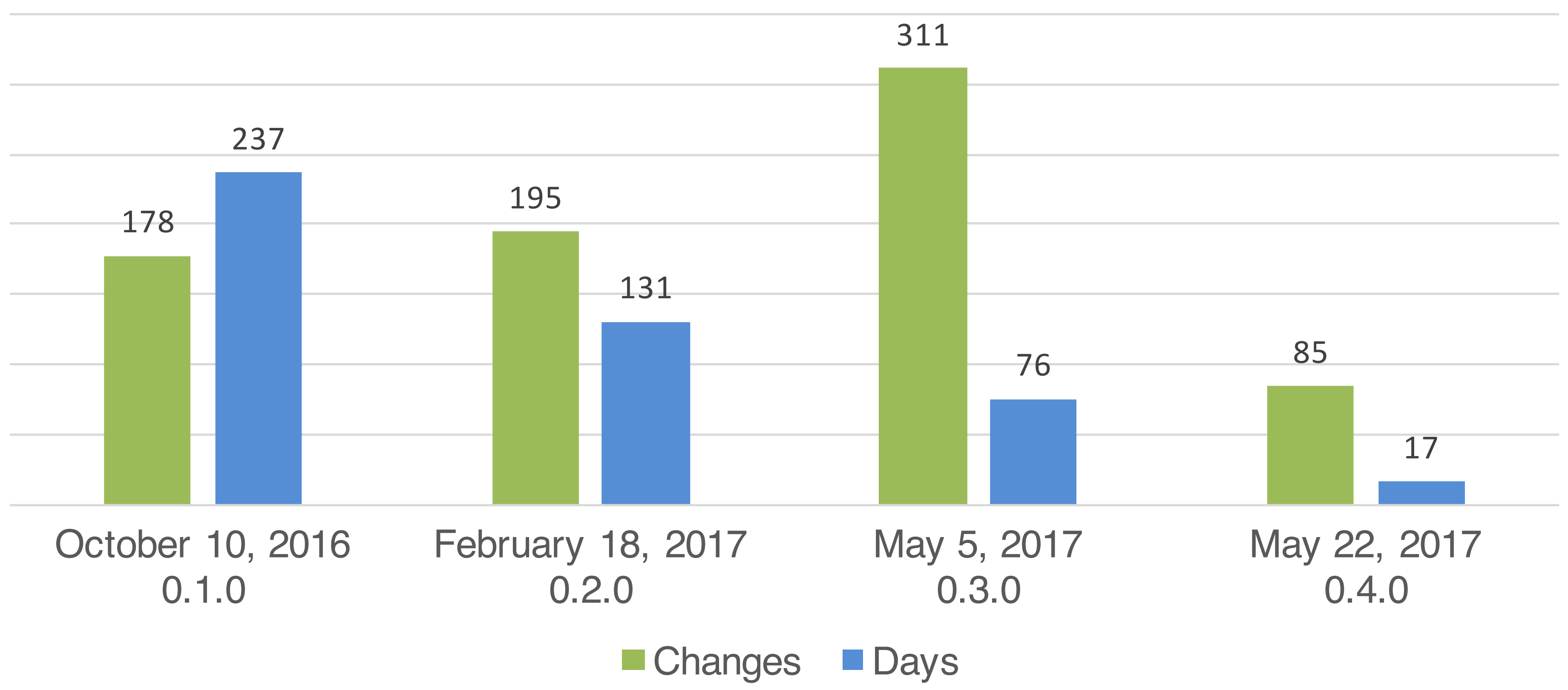
- PySpark Integration:
  53x speedup (IBM spark work on SPARK-13534)
  http://s.apache.org/arrowresult1

- Streaming Arrow Performance
  7.75GB/s data movement
  http://s.apache.org/arrowresult2

- Arrow Parquet C++ Integration
  4GB/s reads
  http://s.apache.org/arrowresult3

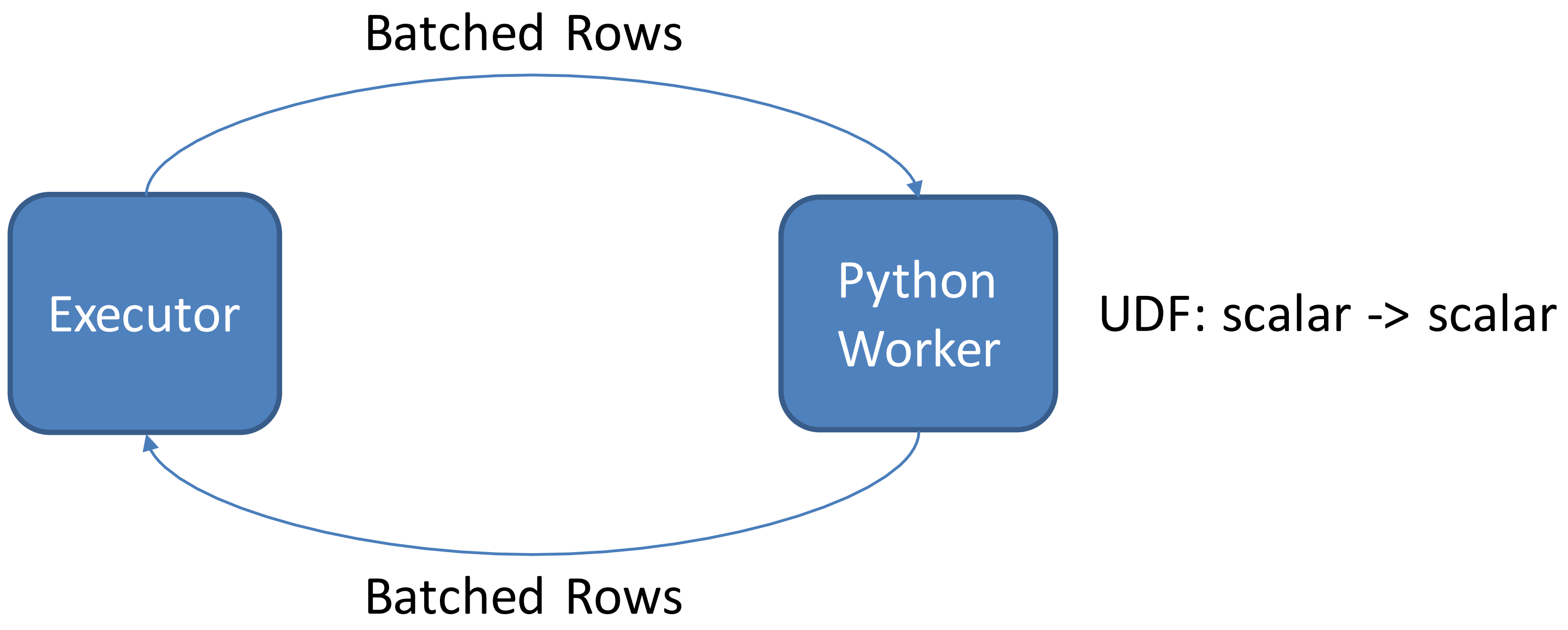
- Pandas Integration
  9.71GB/s
  http://s.apache.org/arrowresult4

Arrow Releases



Improvements to PySpark with Arrow

How PySpark UDF works



Current Issues with UDF

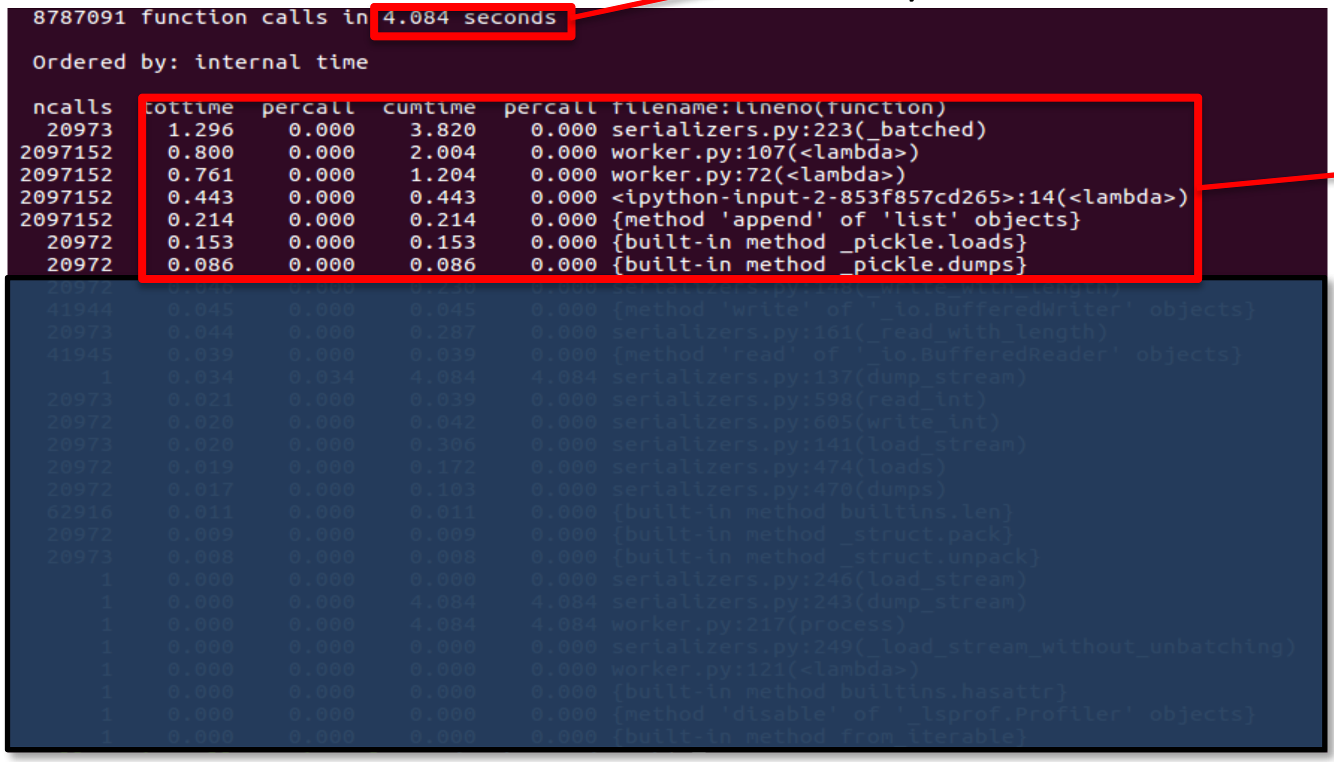
- Serialize / Deserialize in Python
- Scalar computation model (Python for loop)

Profile lambda x: x+1

Actual Runtime is **2s** without profiling.

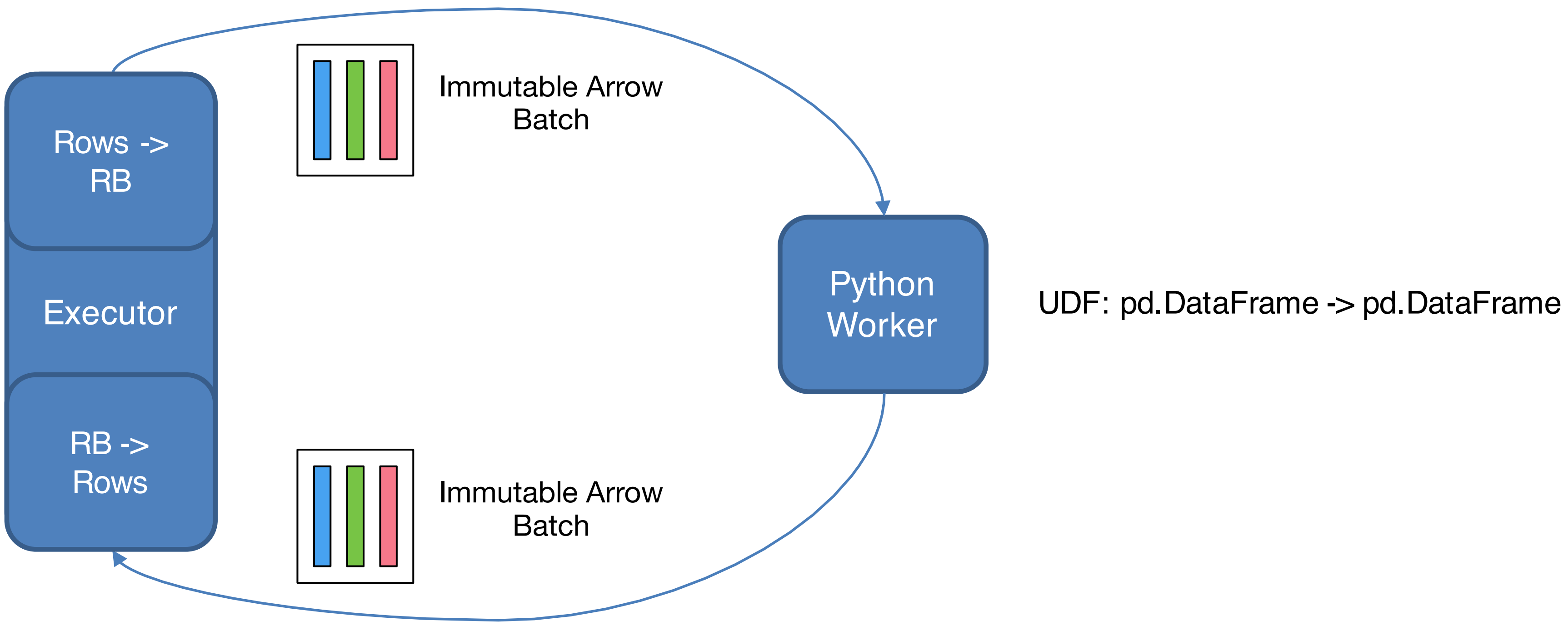
8 Mb/s

91.8%



```
 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}
```

Vectorize Row UDF

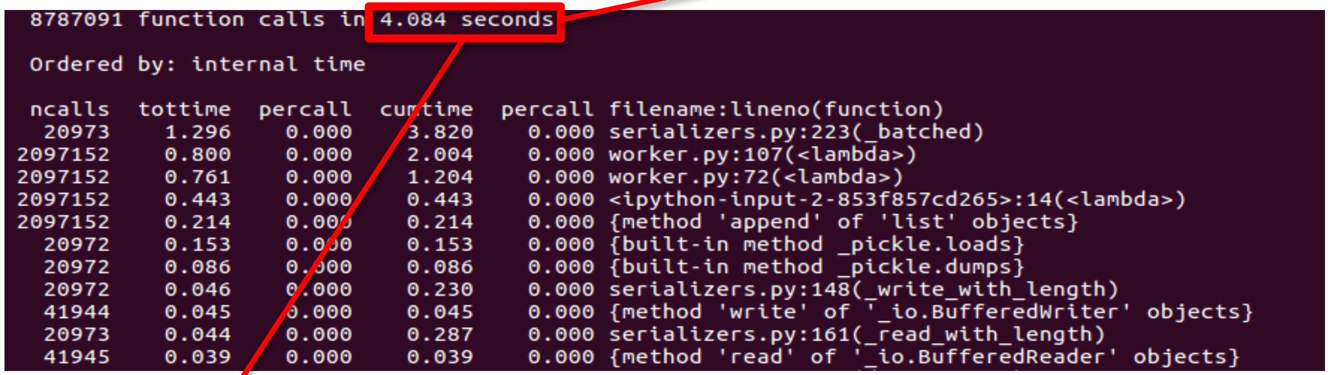


Why pandas.DataFrame

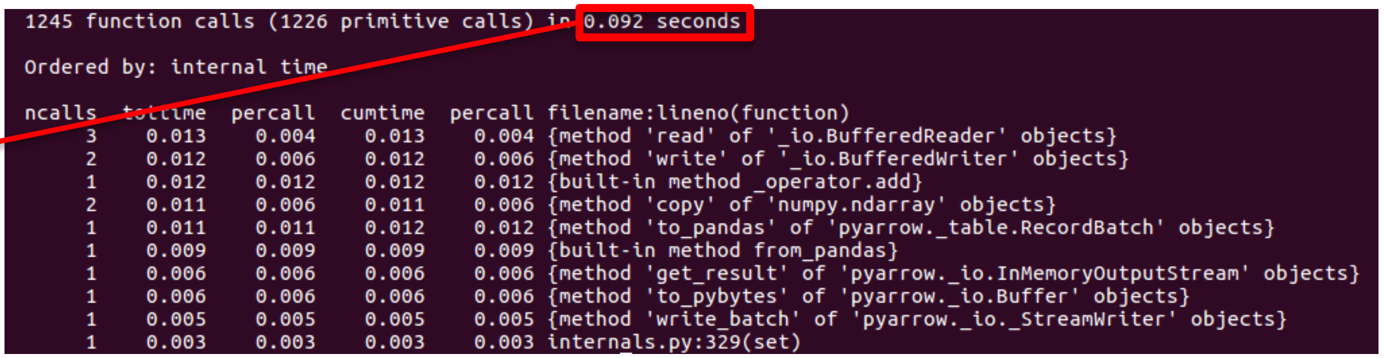
- Fast, feature-rich, widely used by Python users
- Already exists in PySpark (toPandas)
- Compatible with popular Python libraries:
  - NumPy, StatsModels, SciPy, scikit-learn…
- Zero copy to/from Arrow

Scalar vs Vectorized UDF

Actual Runtime is **2s** without profiling



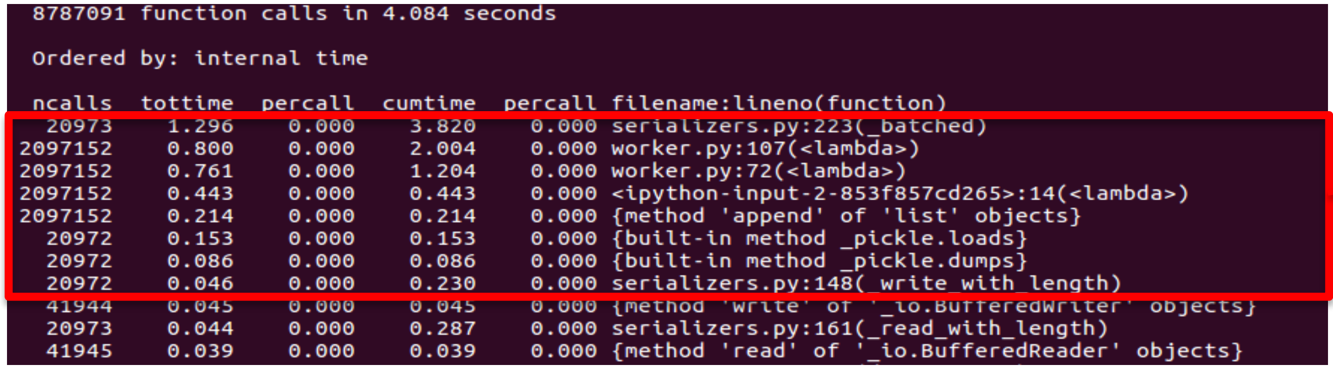
```
 8787091 function calls in 4.084 seconds

 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}
```



```
 1245 function calls (1226 primitive calls) in 0.092 seconds

 Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3    0.013    0.004    0.013    0.004 {method 'read' of '_io.BufferedReader' objects}
      2    0.012    0.006    0.012    0.006 {method 'write' of '_io.BufferedWriter' objects}
      1    0.012    0.012    0.012    0.012 {built-in method _operator.add}
      2    0.011    0.006    0.011    0.006 {method 'copy' of 'numpy.ndarray' objects}
      1    0.011    0.011    0.012    0.012 {method 'to_pandas' of 'pyarrow._table.RecordBatch' objects}
      1    0.009    0.009    0.009    0.009 {built-in method from_pandas}
      1    0.006    0.006    0.006    0.006 {method 'get_result' of 'pyarrow._io.InMemoryOutputStream' objects}
      1    0.006    0.006    0.006    0.006 {method 'to_pybytes' of 'pyarrow._io.Buffer' objects}
      1    0.005    0.005    0.005    0.005 {method 'write_batch' of 'pyarrow._io._StreamWriter' objects}
      1    0.003    0.003    0.003    0.003 internals.py:329(set)
```

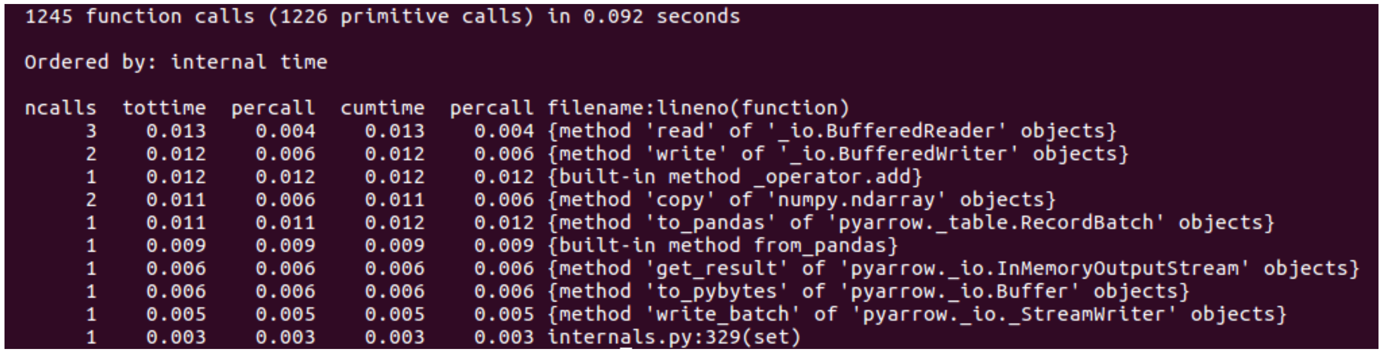
20x Speed Up

Scalar vs Vectorized UDF



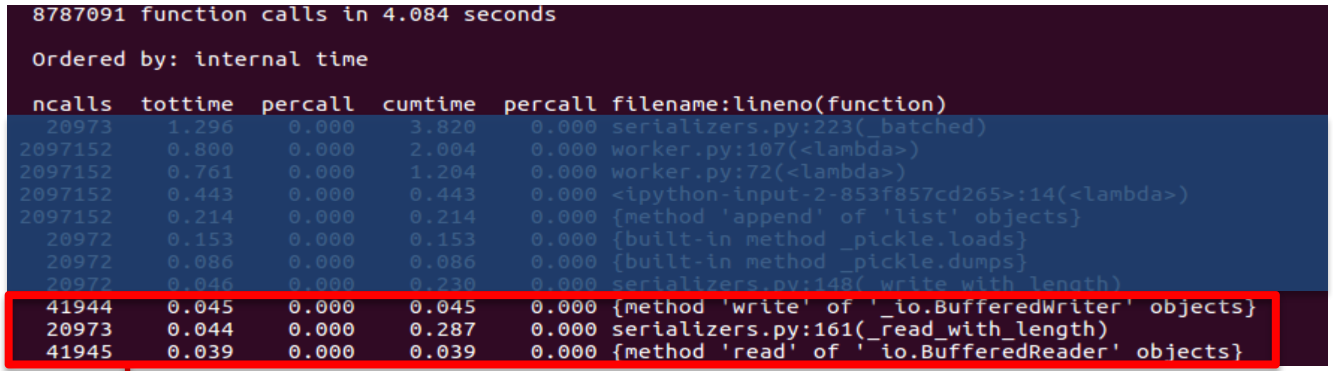
```
 8787091 function calls in 4.084 seconds

 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}
```

Overhead
Removed

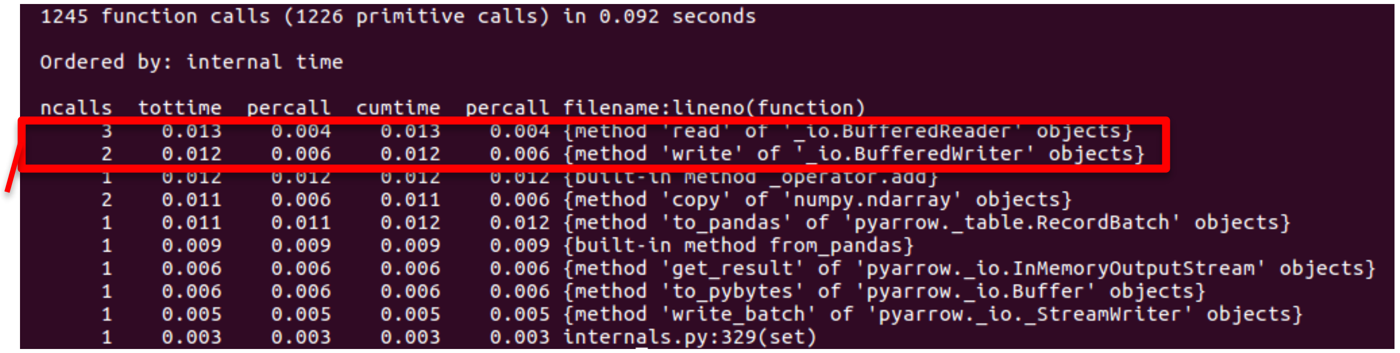


```
1245 function calls (1226 primitive calls) in 0.092 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     3    0.013    0.004    0.013    0.004 {method 'read' of '_io.BufferedReader' objects}
     2    0.012    0.006    0.012    0.006 {method 'write' of '_io.BufferedWriter' objects}
     1    0.012    0.012    0.012    0.012 {built-in method _operator.add}
     2    0.011    0.006    0.011    0.006 {method 'copy' of 'numpy.ndarray' objects}
     1    0.011    0.011    0.012    0.012 {method 'to_pandas' of 'pyarrow._table.RecordBatch' objects}
     1    0.009    0.009    0.009    0.009 {built-in method from_pandas}
     1    0.006    0.006    0.006    0.006 {method 'get_result' of 'pyarrow._io.InMemoryOutputStream' objects}
     1    0.006    0.006    0.006    0.006 {method 'to_pybytes' of 'pyarrow._io.Buffer' objects}
     1    0.005    0.005    0.005    0.005 {method 'write_batch' of 'pyarrow._io._StreamWriter' objects}
     1    0.003    0.003    0.003    0.003 internals.py:329(set)
```

Scalar vs Vectorized UDF



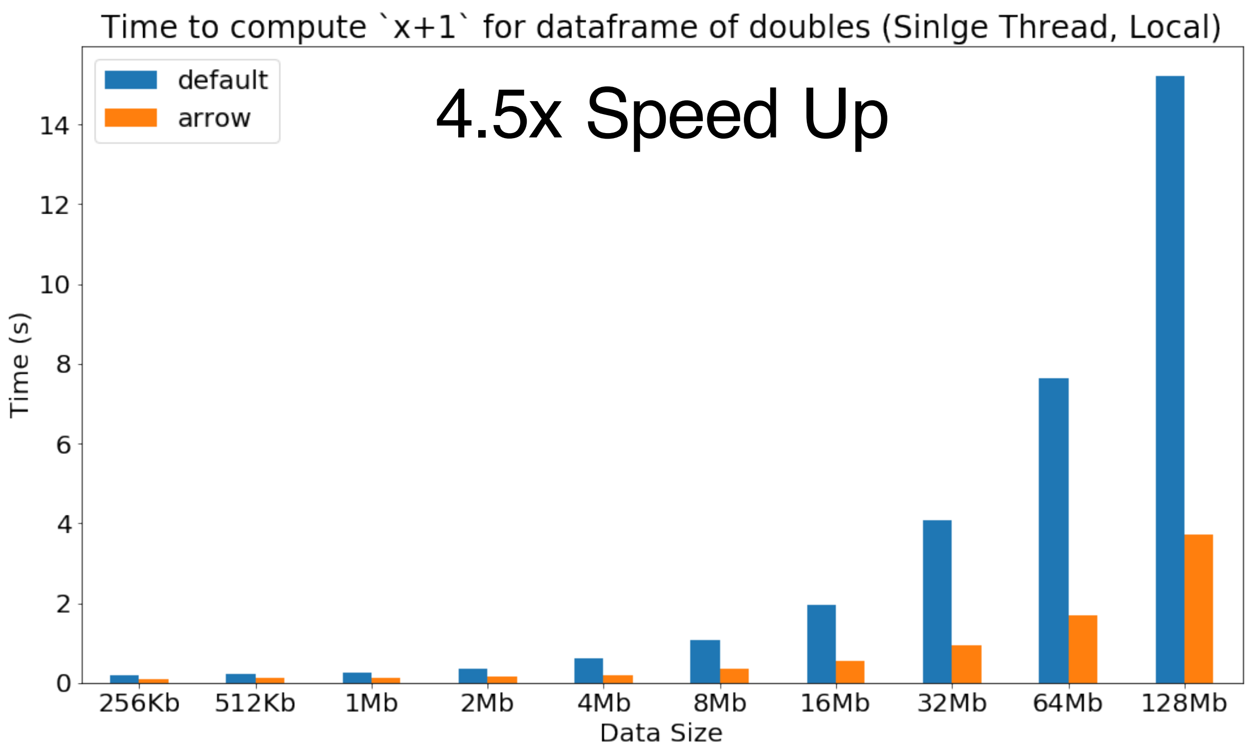
```
 8787091 function calls in 4.084 seconds

 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}
```



```
 1245 function calls (1226 primitive calls) in 0.092 seconds

 Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      3    0.013    0.004    0.013    0.004 {method 'read' of '_io.BufferedReader' objects}
      2    0.012    0.006    0.012    0.006 {method 'write' of '_io.BufferedWriter' objects}
      1    0.012    0.012    0.012    0.012 {built-in method _operator.add}
      2    0.011    0.006    0.011    0.006 {method 'copy' of 'numpy.ndarray' objects}
      1    0.011    0.011    0.012    0.012 {method 'to_pandas' of 'pyarrow._table.RecordBatch' objects}
      1    0.009    0.009    0.009    0.009 {built-in method from_pandas}
      1    0.006    0.006    0.006    0.006 {method 'get_result' of 'pyarrow._io.InMemoryOutputStream' objects}
      1    0.006    0.006    0.006    0.006 {method 'to_pybytes' of 'pyarrow._io.Buffer' objects}
      1    0.005    0.005    0.005    0.005 {method 'write_batch' of 'pyarrow._io._StreamWriter' objects}
      1    0.003    0.003    0.003    0.003 internals.py:329(set)
```

Less System Call
Faster I/O

Scalar vs Vectorized UDF



Support Group UDF

- Split-apply-combine:
  - Break a problem into smaller pieces
  - Operate on each piece independently
  - Put all pieces back together
- Common pattern supported in SQL, Spark, Pandas, R …

Split-Apply-Combine (Current)

- Split: groupBy, window, …
- Apply: mean, stddev, collect_list, rank …
- Combine: Inherently done by Spark

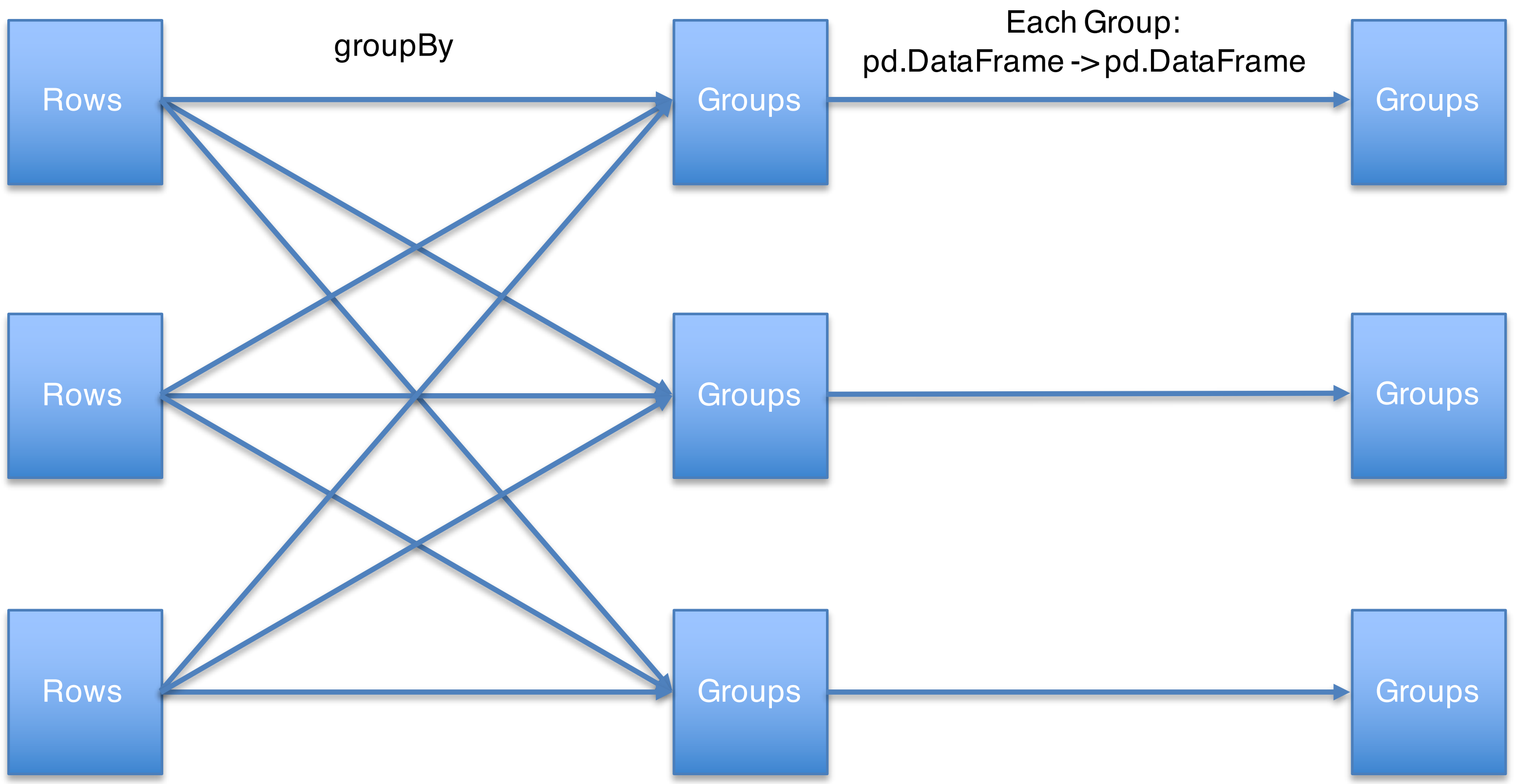
Split-Apply-Combine (with Group UDF)

- Split: groupBy, window, …
- Apply: UDF
- Combine: Inherently done by Spark

Introduce groupBy().apply()

- UDF: pd.DataFrame -> pd.DataFrame
  - Treat each group as a pandas DataFrame
  - Apply UDF on each group
  - Assemble as PySpark DataFrame

Introduce groupBy().apply()



Previous Example: Data Normalization

(values - values.mean()) / values.std()

Previous Example: Data Normalization

Current:

```
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')
```

Group UDF:

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

def normalize(df):
    v1 = df.v1
    df['v3'] = (v1 - v1.mean()) / v1.std()
    return df

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

5x Speed Up

Limitations

- Requires Spark Row <-> Arrow RecordBatch conversion
  - Incompatible memory layout (row vs column)
- (groupBy) No local aggregation
  - Difficult due to how PySpark works. See https://issues.apache.org/jira/browse/SPARK-10915

Future Roadmap

What’s Next (Arrow)

- Arrow RPC/REST
- Arrow IPC
- Apache {Spark, Drill, Kudu} to Arrow Integration
  - Faster UDFs, Storage interfaces

What’s Next (PySpark UDF)

- Continue working on SPARK-20396
- Support Pandas UDF with more PySpark functions:
  - groupBy().agg()
  - window

What’s Next (PySpark UDF)

```
import numpy as np
@pandas_udf(Scalar, DoubleType())
def weighted_mean_udf(v1, w):
    return np.average(v1, weights=w)

df.groupBy('id').agg(weighted_mean_udf(df.v1, df.w).as('v1_wm'))
```

```
w = Window.partitionBy('id')

@pandas_udf(Series, DoubleType())
def rank_udf(v):
    return v.rank(pct=True)

df.withColumn('rank', rank_udf(df.v).over(w))
```

Get Involved

- Watch SPARK-20396
- Join the Arrow community
  - dev@arrow.apache.org
  - Slack:
    - https://apachearrowslackin.herokuapp.com/
  - http://arrow.apache.org
  - Follow @ApacheArrow

Thank you

- Bryan Cutler (IBM), Wes McKinney (Two Sigma Investments) for helping build this feature
- Apache Arrow community
- Spark Summit organizers
- Two Sigma and Dremio for supporting this work