TWO SIGMA

www.twosigma.com

Improving Python and Spark Performance and Interoperability

Wes McKinney @wesmckinn

Spark Summit East 2017

February 9, 2017

Me

- Currently: Software Architect at Two Sigma Investments
- Creator of Python pandas project
- PMC member for Apache Arrow and Apache Parquet
- Other Python projects: Ibis, Feather, statsmodels
- Formerly: Cloudera, DataPad, AQR
- Author of *Python for Data Analysis*

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.

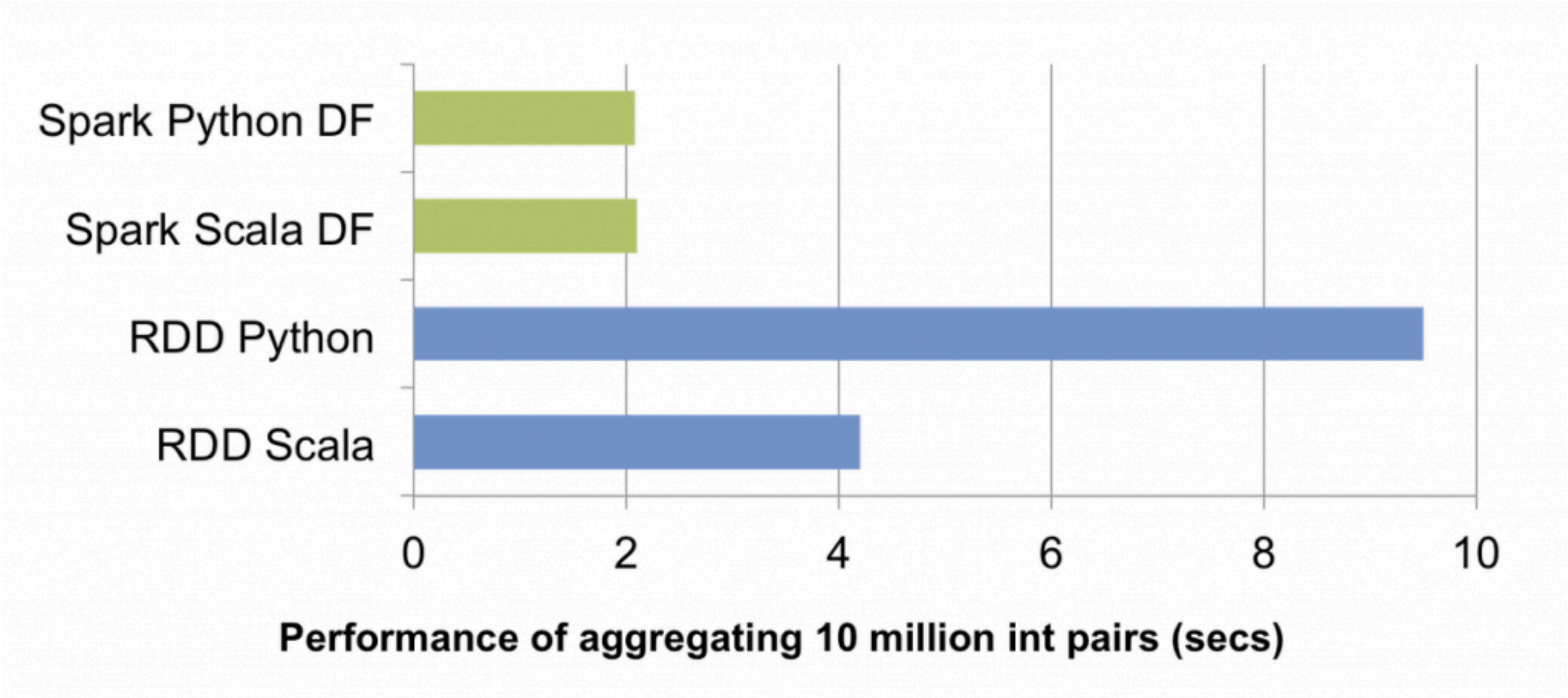
This talk

- Why some parts of PySpark are “slow”
- Technology that can help make things faster
- Work we have done to make improvements
- Future roadmap

Python and Spark

- Spark is implemented in Scala, runs on the Java virtual machine (JVM)
- Spark has Python and R APIs with partial or full coverage for many parts of the Scala Spark API
- In some Spark tasks, **Python is only a scripting front-end.**
  - **This means no interpreted Python code is executed once the Spark job starts**
- Other PySpark jobs suffer performance and interoperability issues that we're going to analyze in this talk

Spark DataFrame performance



Source: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

Spark DataFrame performance can be misleading

- Spark DataFrames are an example of Python as a DSL / scripting front end
- Excepting UDFs (.map(…) or sqlContext.registerFunction), no Python code is evaluated in the Spark job
- Python API calls create SQL query plans inside the JVM — so Scala and Python versions are computationally identical

Spark DataFrames as deferred DSL

```
young = users[users.age < 21]
young.groupBy(“gender”).count()
```

Spark DataFrames as deferred DSL

```
SELECT gender, COUNT(*)
FROM users
WHERE age < 21
GROUP BY 1
```

Spark DataFrames as deferred DSL

```
Aggregation[table]
  table:
    Table: users
  metrics:
    count = Count[int64]
      Table: ref_0
  by:
    gender = Column[array(string)] 'gender' from users
  predicates:
    Less[array(boolean)]
      age = Column[array(int32)] 'age' from users
      Literal[int8]
        21
```

Where Python code and Spark meet

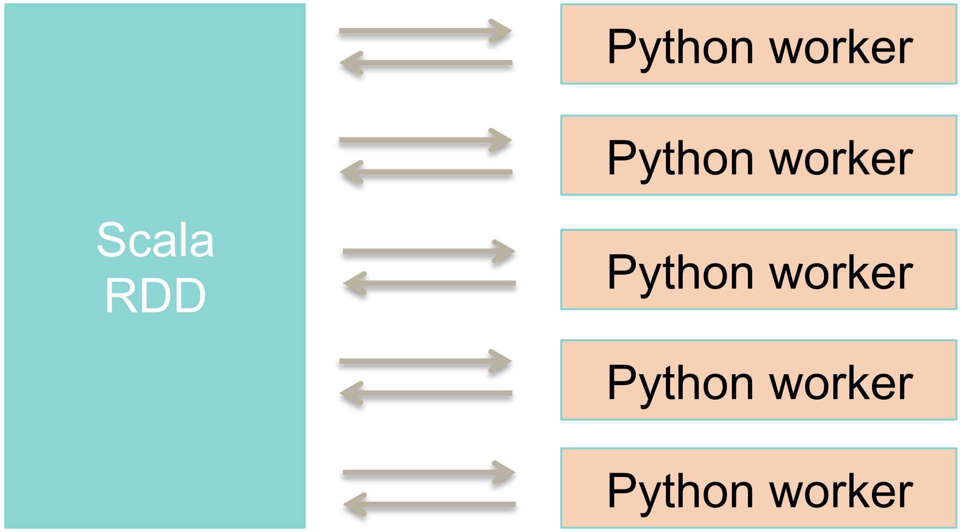
- Unfortunately, many PySpark jobs cannot be expressed entirely as DataFrame operations or other built-in Scala constructs

- Spark-Scala interacts with in-memory Python in key ways:
  - **Reading and writing in-memory datasets to/from the Spark driver**
  - **Evaluating custom Python code (user-defined functions)**

How PySpark lambda functions work

- The anatomy of

```
rdd.map(lambda x: … )
df.withColumn(py_func(...))
```



see PythonRDD.scala

PySpark lambda performance problems

- See 2016 talk “High Performance Python on Apache Spark”
  - http://www.slideshare.net/wesm/high-performance-python-on-apache-spark
- Problems
  - Inefficient data movement (serialization / deserialization)
  - Scalar computation model: object boxing and interpreter overhead
- General summary: PySpark is not currently designed to achieve high performance in the way that pandas and NumPy are.

Other issues with PySpark lambdas

- Computation model unlike what pandas users are used to
  - In `dataframe.map(f)`, the Python function `f` only sees one Row at a time
- A more natural and efficient vectorized API would be:
  - `dataframe.map_pandas(lambda df: …)`

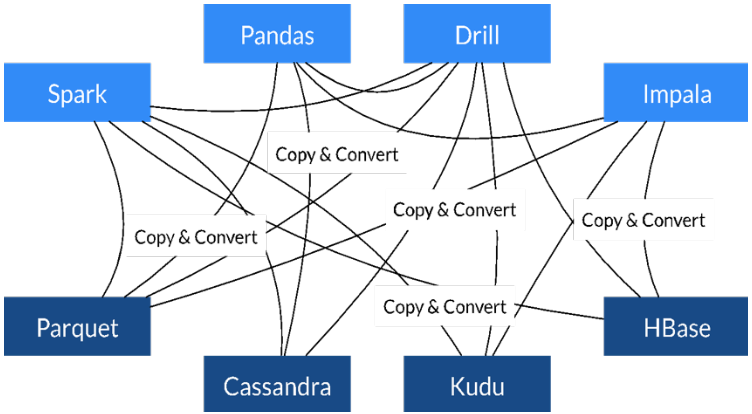
Apache Arrow

Apache Arrow: Process and Move Data Fast

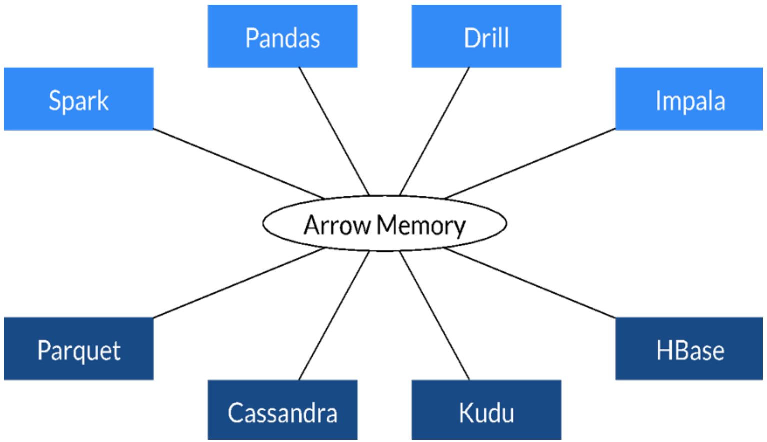
- New Top-level Apache project as of February 2016
- Collaboration amongst broad set of OSS projects around shared needs
- Language-independent columnar data structures
- Metadata for describing schemas / chunks of data
- Protocol for moving data between processes with minimal serialization overhead

High performance data interchange

Today



With Arrow

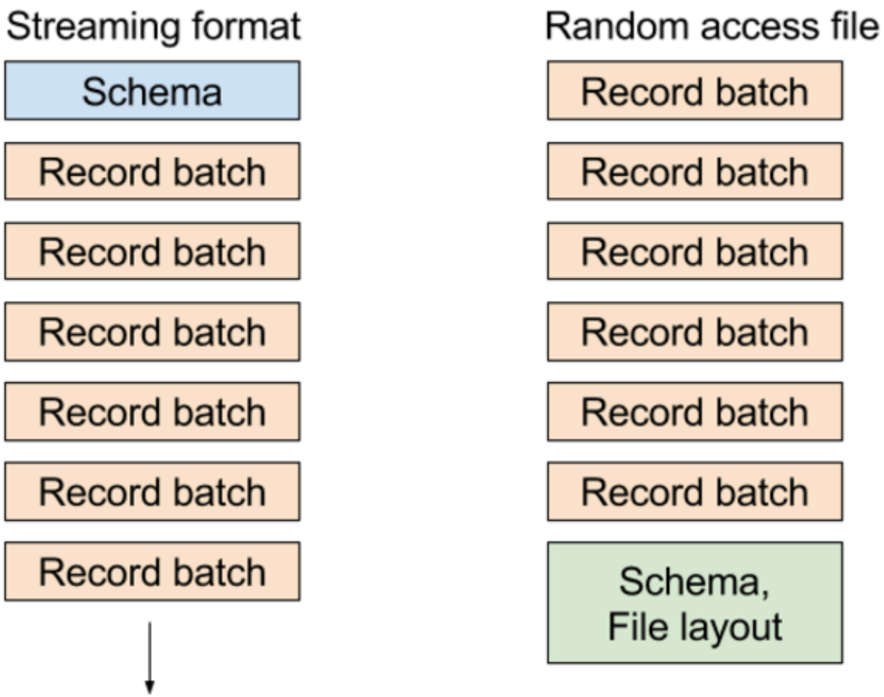


Source: Apache Arrow

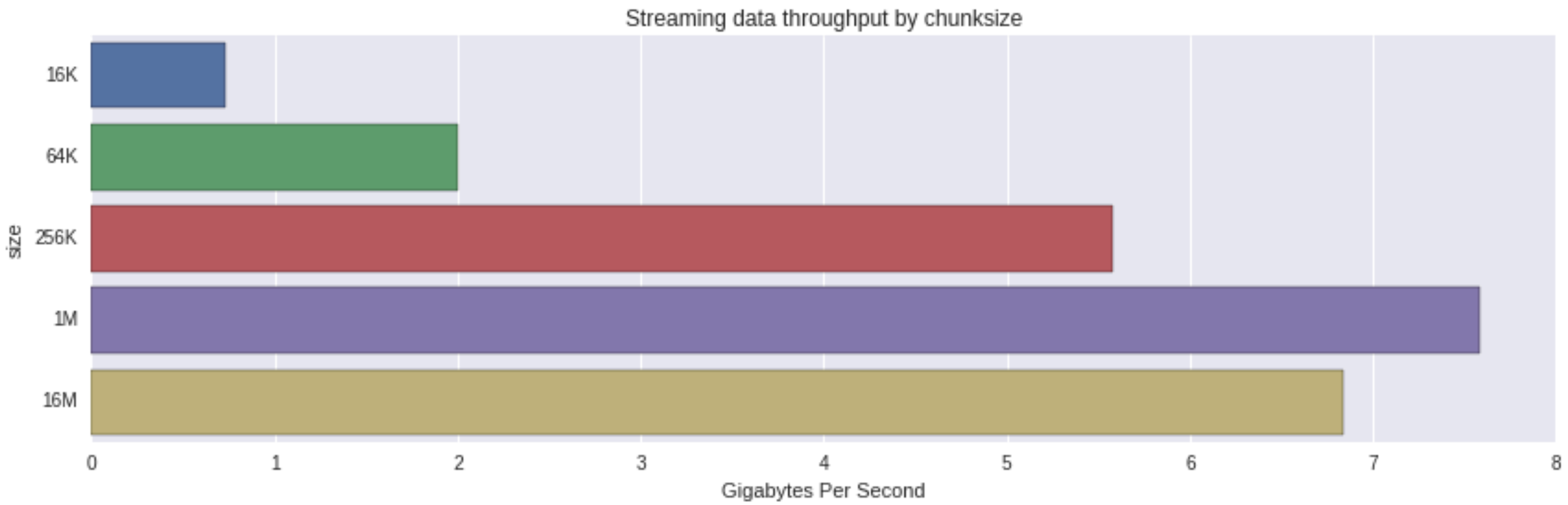
What does Apache Arrow give you?

- **Zero-copy columnar data:** Complex table and array data structures that can reference memory without copying it
- **Ultrafast messaging:** Language-agnostic metadata, batch/file-based and streaming binary formats
- **Complex schema support**: Flat and nested data types
- **C++, Python, and Java Implementations**: with integration tests

Arrow binary wire formats



Extreme performance to pandas from Arrow streams



PyArrow file and streaming API

```
from pyarrow import StreamReader

reader = StreamReader(stream)

# pyarrow.Table
table = reader.read_all()

# Convert to pandas
df = table.to_pandas()
```

Making DataFrame.toPandas faster

- Background
  - Spark's toPandas transfers in-memory from the Spark driver to Python and converts it to a pandas.DataFrame. It is very slow
  - Joint work with Bryan Cutler (IBM), Li Jin (Two Sigma), and Yin Xusen (IBM). See SPARK-13534 on JIRA

- Test case: transfer 128MB Parquet file with 8 DOUBLE columns

```
conda install pyarrow -c conda-forge
```

Making DataFrame.toPandas faster

```
df = sqlContext.read.parquet('example2.parquet')
df = df.cache()
df.count()
```

Then

```
%%prun -s cumulative
dfs = [df.toPandas() for i in range(5)]
```


Making DataFrame.toPandas faster

```
94483943 function calls (94478223 primitive calls) in 62.492 seconds

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       5    1.458    0.292   62.492   12.498 dataframe.py:1570(toPandas)
       5    0.661    0.132   54.759   10.952 dataframe.py:382(collect)
10485765    0.669    0.000   46.823    0.000 rdd.py:121(_load_from_socket)
     715    0.002    0.000   46.139    0.065 serializers.py:141(load_stream)
     710    0.002    0.000   45.950    0.065 serializers.py:448(loads)
10485760    4.969    0.000   32.853    0.000 types.py:595(fromInternal)
    1391    0.004    0.000    7.445    0.005 socket.py:562(readinto)
      18    0.000    0.000    7.283    0.405 java_gateway.py:1006(send_command)
       5    0.000    0.000    6.262    1.252 frame.py:943(from_records)
```

Making DataFrame.toPandas faster

Now, using pyarrow

```
%%prun -s cumulative
dfs = [df.toPandas(useArrow) for i in range(5)]
```


Making DataFrame.toPandas faster

```
38585 function calls (38535 primitive calls) in 9.448 seconds

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       5    0.001    0.000    9.448    1.890 dataframe.py:1570(toPandas)
       5    0.000    0.000    9.358    1.872 dataframe.py:394(collectAsArrow)
    6271    9.330    0.001    9.330    0.001 {method 'recv_into' of '_socket.socket‘}
      15    0.000    0.000    9.229    0.615 java_gateway.py:860(send_command)
      10    0.000    0.000    0.123    0.012 serializers.py:141(load_stream)
       5    0.085    0.017    0.089    0.018 {method 'to_pandas' of 'pyarrow.Table‘}
```

pip install memory_profiler

```
%%memit -i 0.0001
pdf = None
pdf = df.toPandas()
gc.collect()

peak memory: 1223.16 MiB,
increment: 1018.20 MiB
```

Plot thickens: memory use

```
%%memit -i 0.0001
pdf = None
pdf = df.toPandas(useArrow=True)
gc.collect()

peak memory: 334.08 MiB,
increment: 258.31 MiB
```

Summary of results

- Current version: average 12.5s (10.2 MB/s)
  - **Deseralization accounts for 88% of time**; the rest is waiting for Spark to send the data
  - Peak memory use 8x (~1GB) the size of the dataset
- Arrow version
  - Average wall clock time of 1.89s (6.61x faster, 67.7 MB/s)
  - **Deserialization accounts for 1% of total time**
  - Peak memory use 2x the size of the dataset (1 memory doubling)
  - Time for Spark to send data 25% higher (1866ms vs 1488 ms)

Aside: reading Parquet directly in Python

```
import pyarrow.parquet as pq


%%timeit
df = pq.read_table('example2.parquet').to_pandas()
10 loops, best of 3: 175 ms per loop
```

Digging deeper

- Why does it take Spark ~1.8 seconds to send 128MB of data over the wire?

```
val collectedRows = queryExecution.executedPlan.executeCollect()
cnvtr.internalRowsToPayload(collectedRows, this.schema)
```

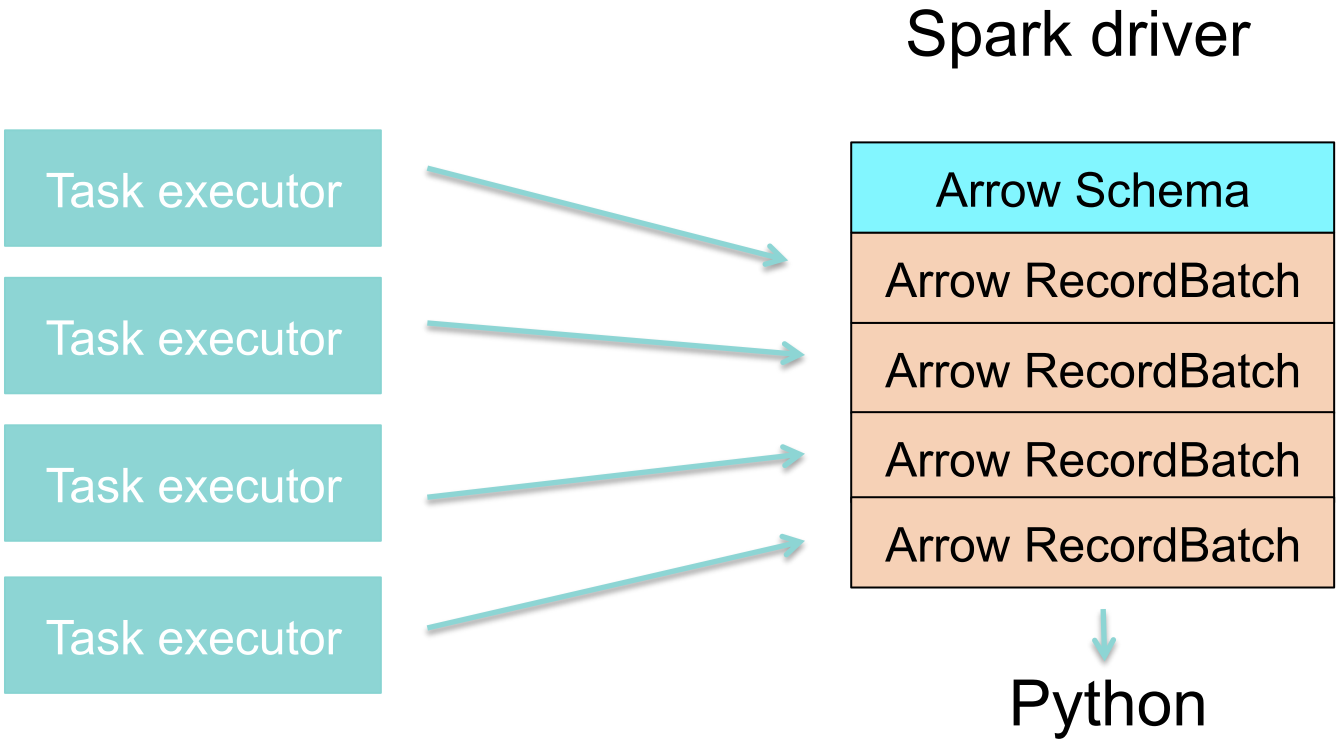
`Array[InternalRow]`

Digging deeper

- In our 128MB test case, on average:
  - **75%** of time is being spent collecting **`Array[InternalRow]`** from the task executors
  - **25%** of the time is spent on a **single-threaded** conversion of all the data from **`Array[InternalRow]`** to **`ArrowRecordBatch`**

- **We can go much faster by performing the Spark SQL -> Arrow conversion locally on the task executors, then streaming the batches to Python**

Future architecture



Hot off the presses

```
 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      5    0.000    0.000    5.928    1.186 dataframe.py:1570(toPandas)
      5    0.000    0.000    5.838    1.168 dataframe.py:394(collectAsArrow)
   5919    0.005    0.000    5.824    0.001 socket.py:561(readinto)
   5919    5.809    0.001    5.809    0.001 {method 'recv_into' of '_socket.socket‘}
...
      5    0.086    0.017    0.091    0.018 {method 'to_pandas' of 'pyarrow.Table‘}
```

Patch from February 8: 38% perf improvement

The work ahead

- Luckily, speeding up **toPandas** and speeding up Lambda / UDF functions is architecturally the same type of problem
- Reasonably clear path to making toPandas even faster
- **How can you get involved?**
  - Keep an eye on Spark ASF JIRA
  - Contribute to Apache Arrow (Java, C++, Python, other languages)
  - Join the Arrow and Spark mailing lists

Thank you

- Bryan Cutler, Li Jin, and Yin Xusen, for building the integration Spark-Arrow integration
- Apache Arrow community
- Spark Summit organizers
- Two Sigma and IBM, for supporting this work