cloudera

High Performance Python on Apache Spark

Wes McKinney @wesmckinn

Spark Summit West -- June 7, 2016

Me

- Data Science Tools at Cloudera
- Serial creator of structured data tools / user interfaces
- Wrote bestseller ***Python for Data Analysis*** 2012
  - Working on **expanded and revised 2nd edition**, coming 2017
- Open source projects
  - Python {pandas, Ibis, statsmodels}
  - Apache {Arrow, Parquet, Kudu (incubating)}
- Focused on C++, Python, and Hybrid projects

Agenda

- Why care about Python?
- What does “high performance Python” even mean?
- A modern approach to Python data software
- Spark and Python: performance analysis and development directions

Why care about (C)Python?

- Accessible, “swiss army knife” programming language
- Highly productive for software engineering and data science alike
- Has excelled as the agile “orchestration” or “glue” layer for application business logic
- Easy to interface with C / C++ / Fortran code. Well-designed Python C API

Defining “High Performance Python”

- The end-user workflow involves primarily Python programming; programs can be invoked with “python app_entry_point.py ...”
- The software uses system resources within an acceptable factor of an equivalent program developed completely in Java or C++
  - Preferably 1-5x slower, not 20-50x
- The software is suitable for interactive / exploratory computing on modestly large data sets (= gigabytes) on a single node

Building fast Python software means embracing certain limitations

Having a healthy relationship with the interpreter

- The Python **interpreter** itself is "slow", as compared with hand-coded C or Java
  - Each line of Python code may feature multiple internal C API calls, temporary data structures, etc.

- Python built-in data structures (numbers, strings, tuples, lists, dicts, etc.) have significant memory and performance use overhead

- Threads performing concurrent CPU or IO work must take care not to block other threads

Mantras for great success

- Key question 1: **Am I making the Python interpreter do a lot of work?**
- Key question 2: **Am I blocking other interpreted code from executing?**
- Key question 3: **Am I handling data (memory) in a “good” way?**

Toy example: interpreted vs. compiled code

```
In [15]: N, K = 1000000, 10
         arr = np.tile(np.random.randn(N), K)
```

```
In [36]: def f(x):
             return x * 2

         def foo_interpreted(arr):
             total = 0
             for x in arr:
                 total += f(x)
             return total

         %time sum_interpreted(arr)

         CPU times: user 968 ms, sys: 4 ms, total: 972 ms
         Wall time: 972 ms

Out[36]: 4683.2203252779564
```

Toy example: interpreted vs. compiled code

```
%%cython

from numpy cimport ndarray, float64_t, import_array
import_array()

# cython: boundscheck = False
# cython: wraparound = False

cdef double f(double x):
    return x * 2

def sum_cython(ndarray[float64_t] arr):
    cdef:
        int i, n = len(arr)
        double total = 0

    for i in range(n):
        total += f(arr[i])

    return total
```

```
%timeit sum_cython(arr)
```

```
100 loops, best of 3: 12.5 ms per loop
```

Cython: 78x faster than interpreted

Toy example: interpreted vs. compiled code

NumPy

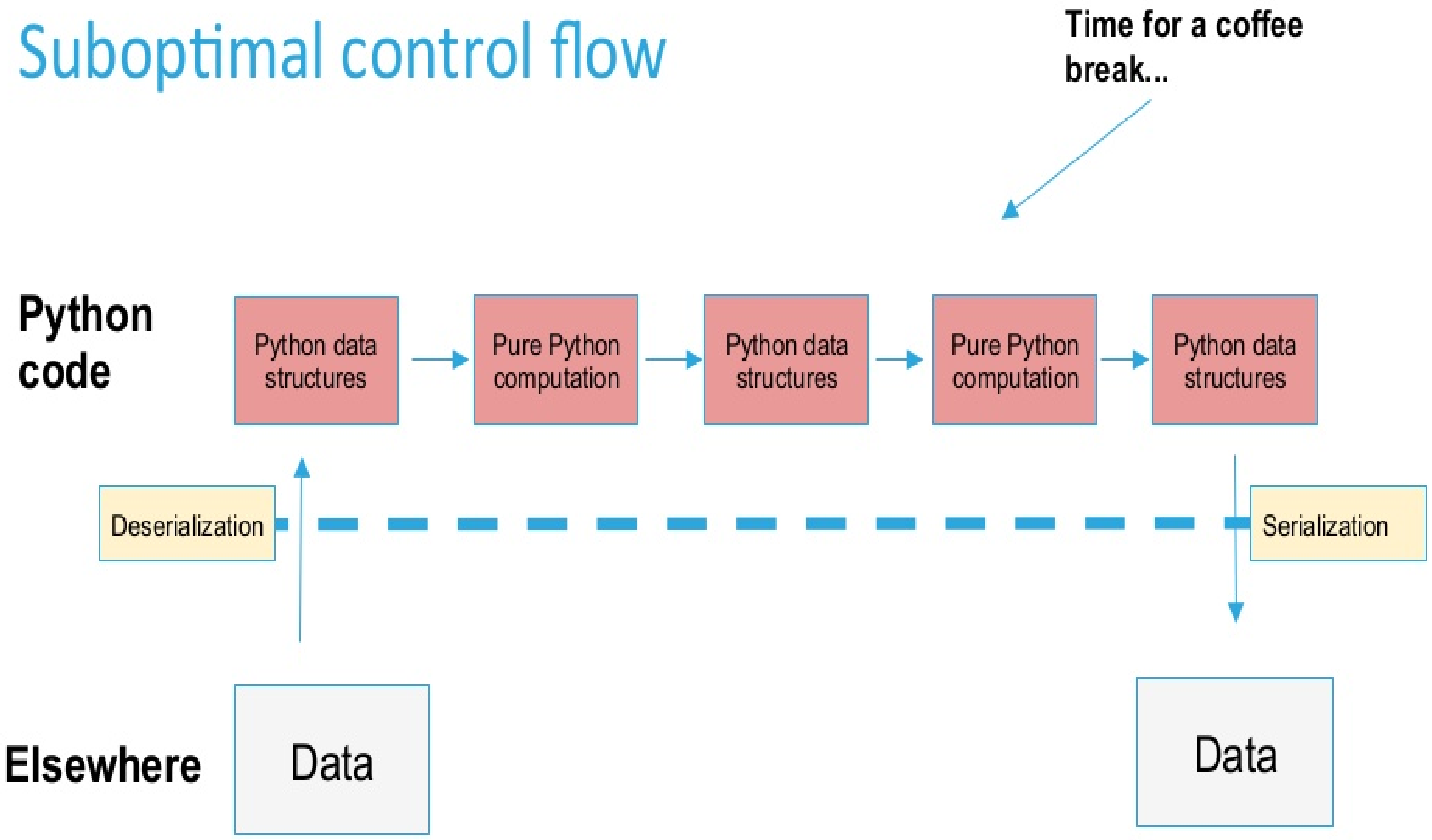
```
In [41]: %timeit f(arr).sum()
         100 loops, best of 3: 17 ms per loop

In [42]: %timeit arr.sum()
         100 loops, best of 3: 6.41 ms per loop
```

Creating a full 80MB temporary array + PyArray_Sum is only 35% slower than a fully inlined Cython (C) function

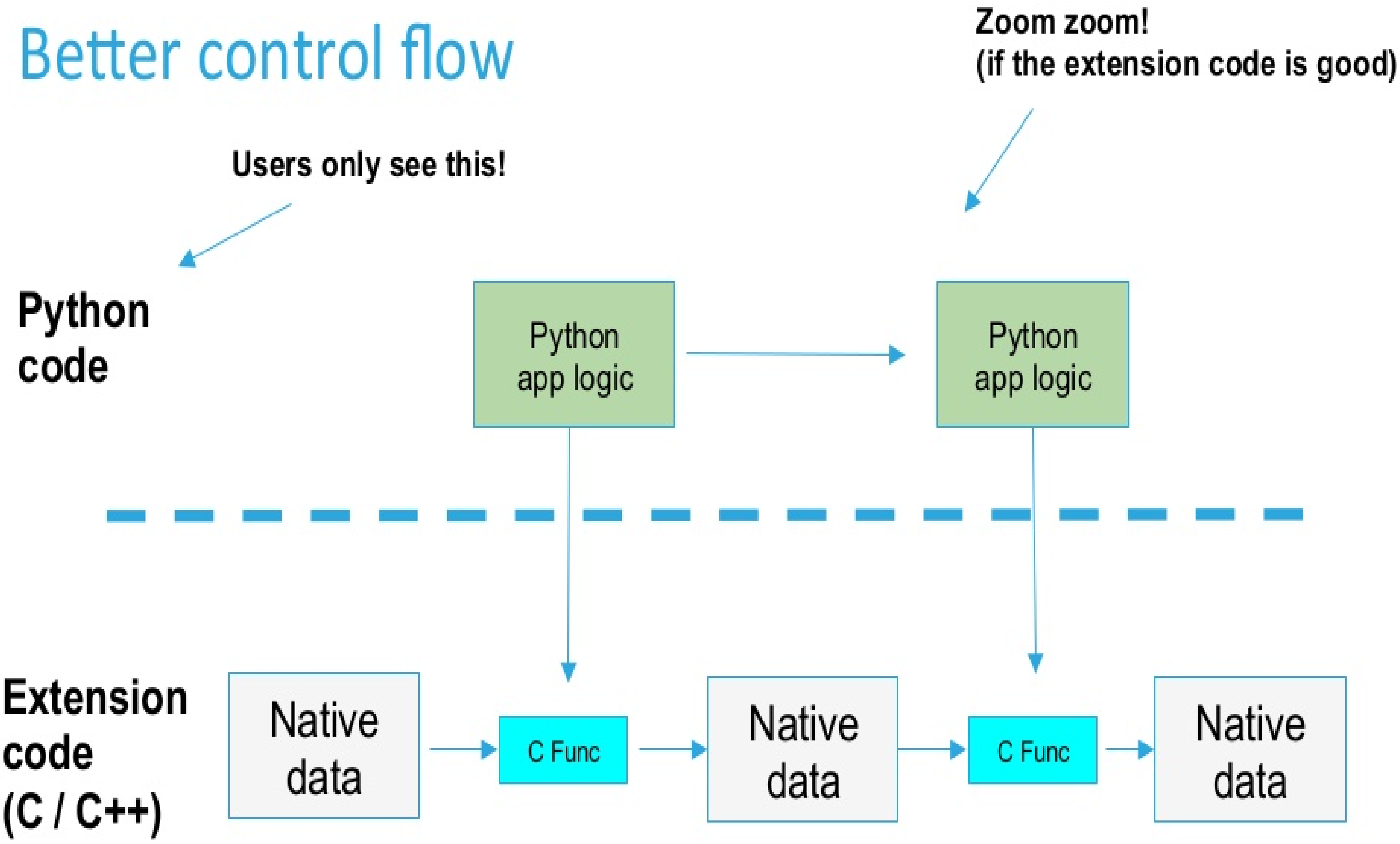
Interesting: ndarray.sum by itself is almost 2x faster than the hand-coded Cython function...

Submarines and Icebergs: metaphors for fast Python software

Suboptimal control flow



Better control flow



But it’s much easier to write 100% Python!

- Building hybrid C/C++ and Python systems adds a lot of complexity to the engineering process
  - (but it’s often worth it)
- See: Cython, SWIG, Boost.Python, Pybind11, and other “hybrid” software creation tools

- **BONUS:** Python programs can orchestrate multi-threaded / concurrent systems written in C/C++ (no Python C API needed)
  - The GIL only comes in when you need to “bubble up” data or control flow (e.g. Python callbacks) into the Python interpreter

A story of reading a CSV file

```
f = get_stream(...)
df = pandas.read_csv(f, **csv_options)
```

Concerns

internally, pseudocode

```
while more_data():
    buffer = f.read()
    parse_bytes(buffer)
df = type_infer_columns()
```

Uses PyString_FromStringAndSize, must hold GIL for this

Synchronous or asynchronous with IO?

Type infer in parallel?
Data structures used?

It's All About the Benjamins (Data Structures)

- The **hard currency** of data software is: **in-memory data structures**
  - How costly are they to send and receive?
  - How costly to manipulate and munge in-memory?
  - How difficult is it to add new proprietary computation logic?

- In Python: **NumPy** established a gold standard for interoperable array data
  - **pandas** is built on NumPy, and made it easy to "plug in" to the ecosystem
    - (but there are plenty of warts still)

What's this have to do with Spark?

- Some known performance issues in PySpark
  - **IO throughput**
    - Python to Spark
    - Spark to Python (or Python extension code)
  - Running interpreted **Python code** on RDDs / Spark DataFrames
    - Lambda mappers / reducers (rdd.map(...))
    - Spark SQL UDFs (registerFunction(...))

Spark IO throughput to/from Python

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.1
      /_/

Using Python version 2.7.11 (default, Dec  6 2015 18:08:32)
SparkContext available as sc, SQLContext available as sqlContext.
```

```
In [11]: N = 1000000
         arr = np.random.randn(N)
         df = pd.DataFrame({'data{0}'.format(i): arr
                            for i in range(10)})

In [12]: %time sdf = sqlContext.createDataFrame(df)

         CPU times: user 1min 5s, sys: 516 ms, total: 1min 6s
         Wall time: 1min 6s

In [13]: %time df2 = sdf.toPandas()

         CPU times: user 4.96 s, sys: 376 ms, total: 5.33 s
         Wall time: 7.77 s
```

Spark 1.6.1 running on localhost

76 MB pandas.DataFrame

1.15 MB/s in

9.82 MB/s out

Spark IO throughput to/from Python

```
def _map_to_pandas(rdds):
    """ Needs to be here due to pickling issues """
    return [pd.DataFrame(list(rdds))]

def toPandas(df, n_partitions=None):
    """
    Returns the contents of `df` as a local `pandas.DataFrame` in 
    repartitioned if `n_partitions` is passed.
    :param df:              pyspark.sql.DataFrame
    :param n_partitions:    int or None
    :return:                pandas.DataFrame
    """
    if n_partitions is not None: df = df.repartition(n_partitions)
    df_pand = df.rdd.mapPartitions(_map_to_pandas).collect()
    df_pand = pd.concat(df_pand)
    df_pand.columns = df.columns
    return df_pand

%time df3 = toPandas(sdf)
```

```
CPU times: user 64 ms, sys: 84 ms, total: 148 ms
Wall time: 2.97 s
```

Unofficial improved toPandas
25.6 MB/s out

Compared with HiveServer2 Thrift RPC fetch

```
In [12]: parquet_table = db.csv_as_parquet
         db = con.database('hs2_perf_test')
         %time df4 = parquet_table.execute(limit=None)

DESCRIBE hs2_perf_test.`csv_as_parquet`
SELECT *
FROM hs2_perf_test.`csv_as_parquet`
CPU times: user 1.04 s, sys: 48 ms, total: 1.08 s
Wall time: 1.84 s

In [7]: import hs2client

In [11]: svc = hs2client.connect('localhost', 21050, 'wesm')
         session = svc.open_session()
         op = session.execute('select * from hs2_perf_test.csv_as_parquet')
         %time df5 = op.fetchall_pandas()

CPU times: user 188 ms, sys: 68 ms, total: 256 ms
Wall time: 840 ms
```

Impala 2.5 + Parquet file on localhost

ibis + impyla 41.46 MB/s read

hs2client (C++ / Python) 90.8 MB/s

Task benchmarked: Thrift TFetchResultsReq + deserialization + conversion to pandas.DataFrame

Back of envelope comp w/ file formats

disclaimer: warm NVMe / OS file cache

```
import feather
```

```
%timeit feather.write_dataframe(df, 'test.feather')
```

```
10 loops, best of 3: 69.1 ms per loop
```

Feather: 1105 MB/s write

```
%timeit feather.read_dataframe('test.feather')
```

```
10 loops, best of 3: 31.6 ms per loop
```

Feather: 2414 MB/s read

```
%time df.to_csv('test.csv', index=False)
```

```
CPU times: user 12.2 s, sys: 144 ms, total: 12.4 s
Wall time: 12.3 s
```

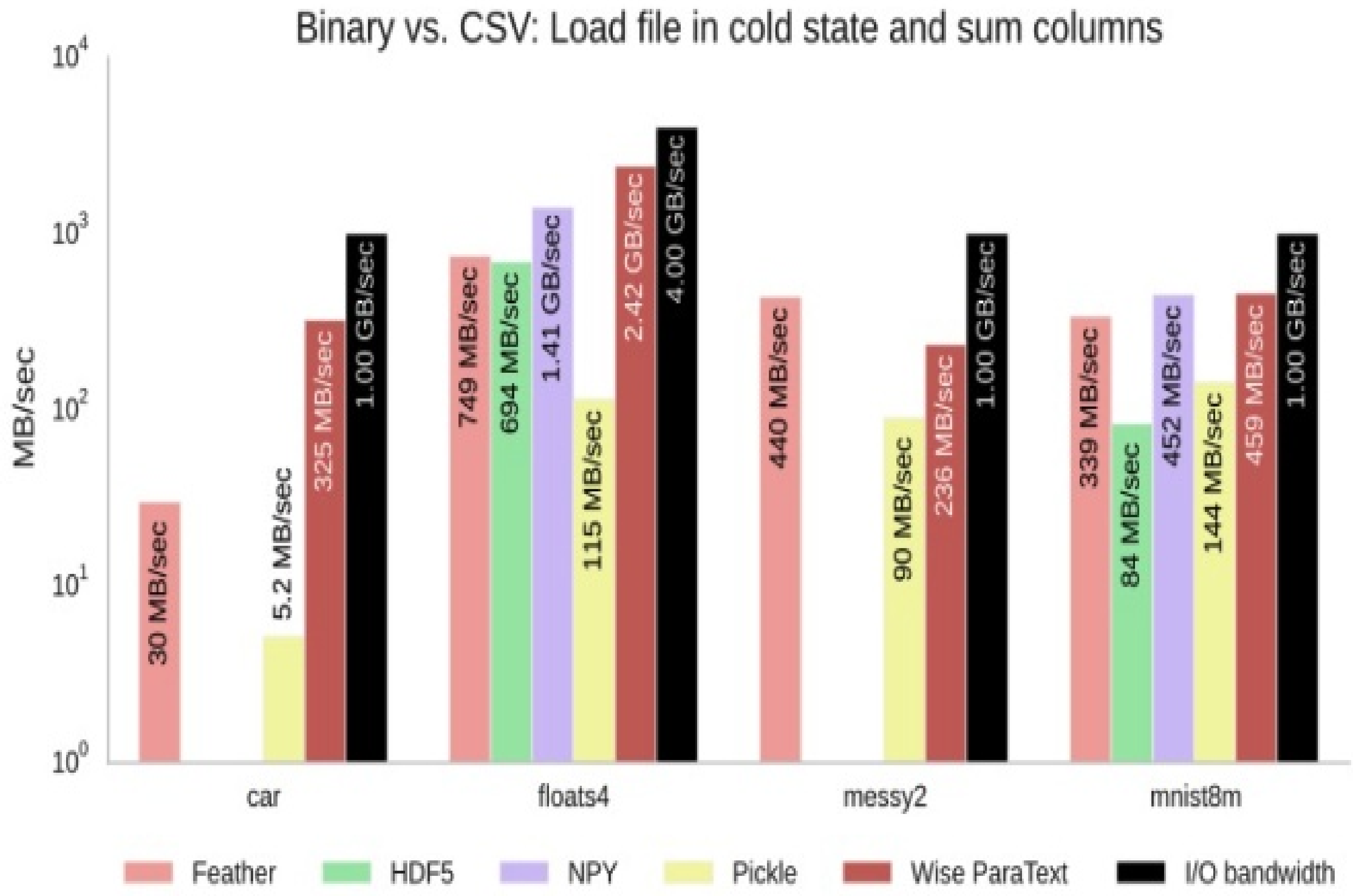
CSV (pandas): 6.2 MB/s write

```
%time df = pd.read_csv('test.csv')
```

```
CPU times: user 1.35 s, sys: 116 ms, total: 1.47 s
Wall time: 1.47 s
```

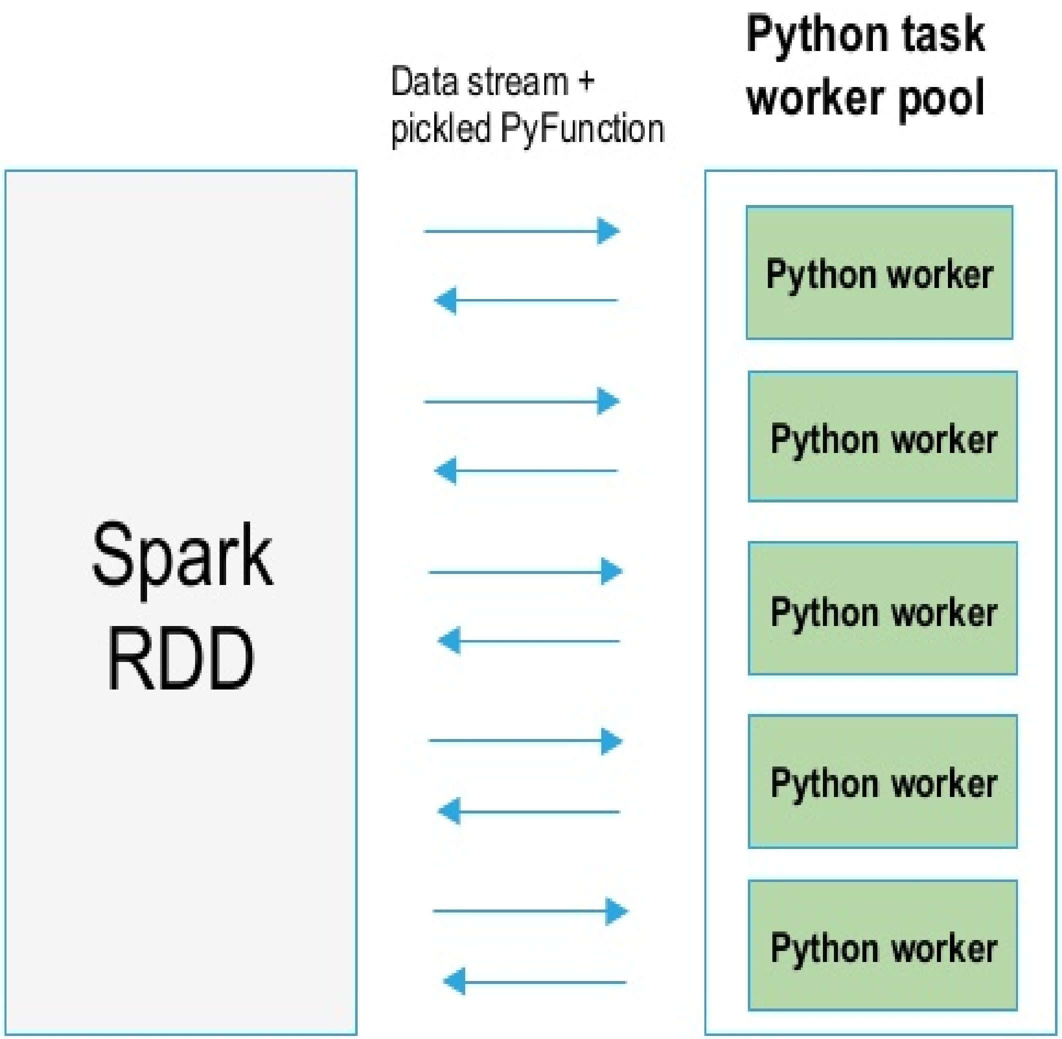
CSV (pandas): 51.9 MB/s read

Aside: CSVs can be fast



See: https://github.com/wiseio/paratext

How Python lambdas work in PySpark



The inner loop of **RDD.map**
map(f, iterator)

See: spark/api/python/PythonRDD.scala
python/pyspark/worker.py

How Python lambdas perform

```
In [44]: rdd = sc.parallelize(arr)

         def f(x):
             return x * 2

         %timeit rdd.map(f).sum()

10 loops, best of 3: 123 ms per loop

In [45]: %timeit (arr * 2).sum()

1000 loops, best of 3: 1.15 ms per loop
```

NumPy array-oriented operations are about 100x faster… but that's not the whole story

Disclaimer: this isn't a remotely "fair" comparison, but it helps illustrate the real pitfalls associated with introducing serialization and RPC/IPC into a computational process

How Python lambdas perform

```
In [44]: rdd = sc.parallelize(arr)

         def f(x):
             return x * 2

         %timeit rdd.map(f).sum()

10 loops, best of 3: 123 ms per loop
```

8 cores

```
In [45]: %timeit (arr * 2).sum()

1000 loops, best of 3: 1.15 ms per loop
```

1 core

Lessons learned: Python data analytics should not be based around scalar object iteration

Asides / counterpoints

- Spark<->Python IO may not be important -- can leave all of the data remote
- Spark DataFrame operations have reduced the need for many types of Lambda functions
- Can use binary file formats as an alternate IO interface
  - Parquet (Python support soon via apache/parquet-cpp)
  - Avro (see cavro, fastavro, pyavroc)
  - ORC (needs a Python champion)
  - ...

Apache Arrow

http://arrow.apache.org

Some slides from Strata-HW talk w/ Jacques Nadeau

Apache Arrow in a Slide

- New Top-level Apache Software Foundation project
  - http://arrow.apache.org

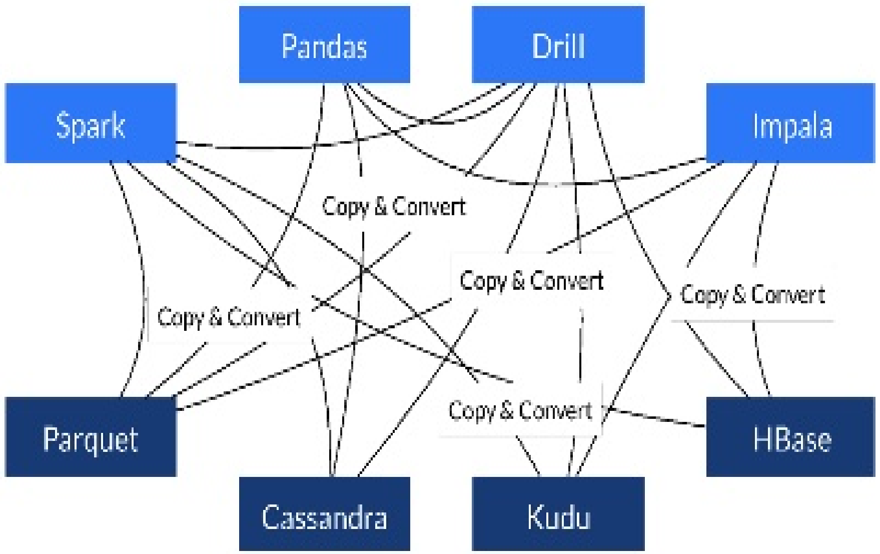
- Focused on Columnar In-Memory Analytics
  1. 10-100x speedup on many workloads
  2. Common data layer enables companies to choose best of breed systems
  3. Designed to work with any programming language
  4. Support for both relational and complex data as-is

- Oriented at collaboration amongst other OSS projects

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

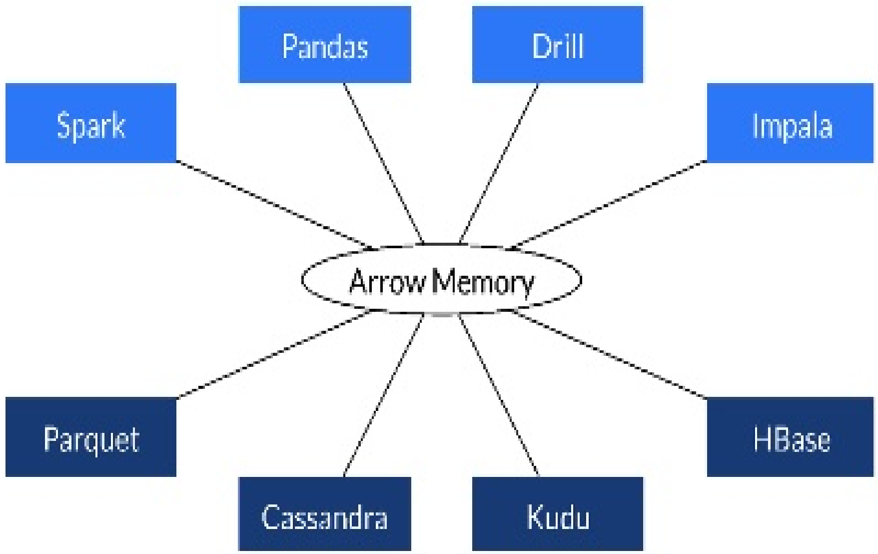
High Performance Sharing & Interchange

Today



- Each system has its own internal memory format
- 70-80% CPU wasted on serialization and deserialization
- Similar functionality implemented in multiple projects

With Arrow



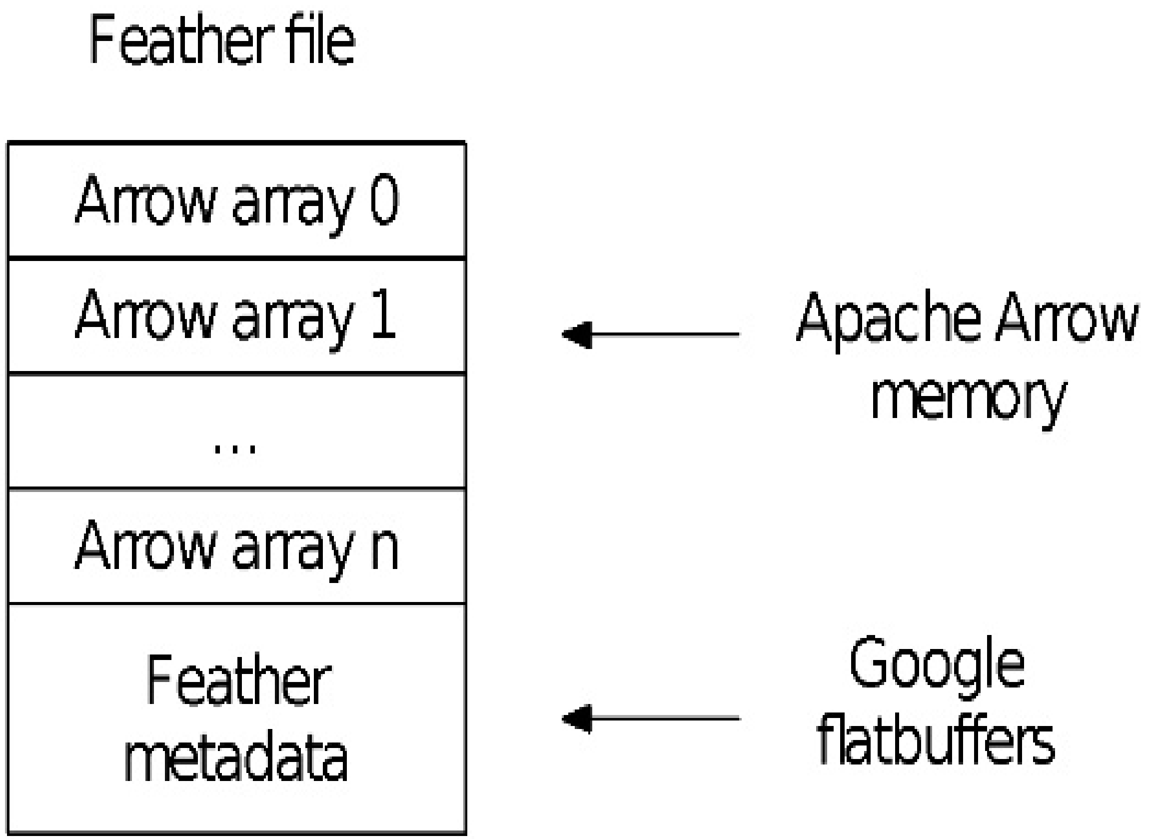
- All systems utilize the same memory format
- No overhead for cross-system communication
- Projects can share functionality (eg, Parquet-to-Arrow reader)

Arrow and PySpark

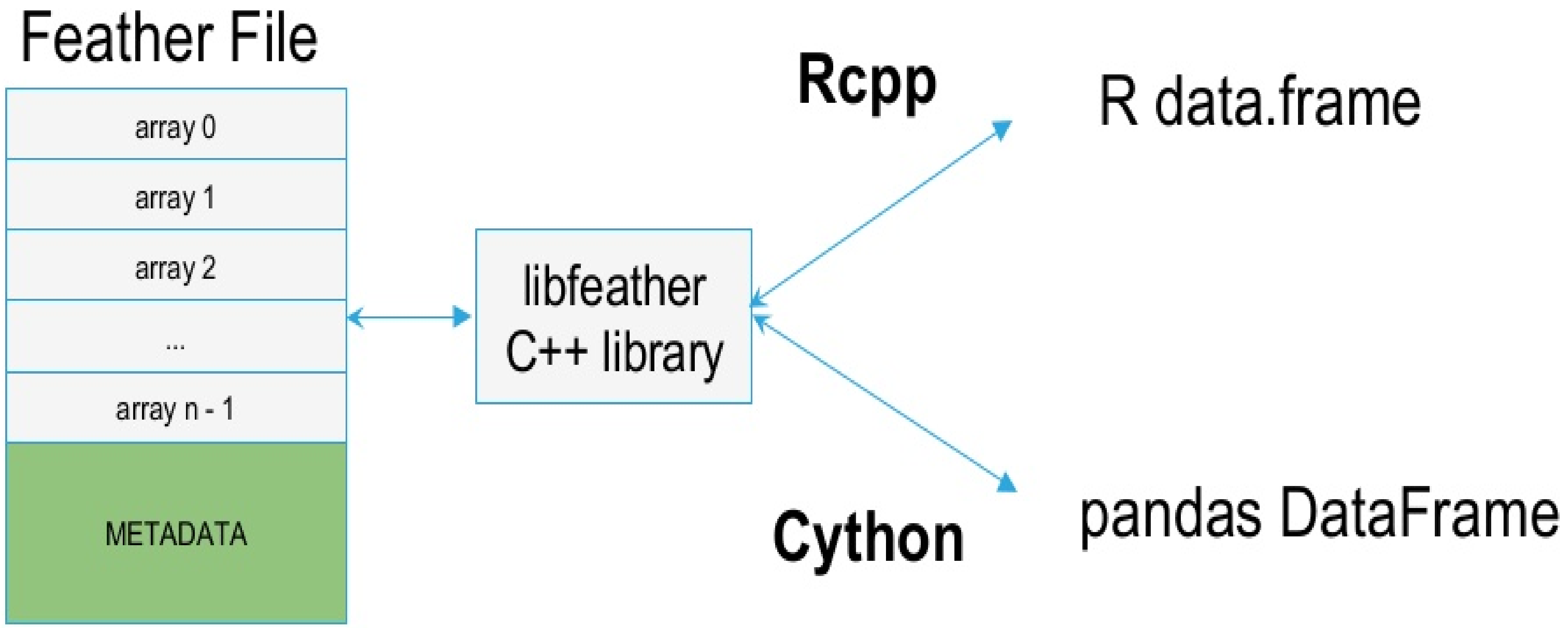
- Build a C API level data protocol to move data between Spark and Python
- Either
  - (Fast) Convert Arrow to/from pandas.DataFrame
  - (Faster) Perform native analytics on Arrow data in-memory
- Use Arrow
  - For efficiently handling nested Spark SQL data in-memory
  - IO: pandas/NumPy data push/pull
  - Lambda/UDF evaluation

Arrow in action: Feather File Format for Python and R

- Problem: fast, language-agnostic binary data frame file format
- Creators: **Wes McKinney** (Python) and **Hadley Wickham** (R)
- Read speeds close to disk IO performance



More on Feather



Summary

- It's essential to improve Spark's low-level data interoperability with the Python data ecosystem

- I'm personally excited to work with the Spark + Arrow + PyData + other communities to help make this a reality cloudera

Thank you

Wes McKinney @wesmckinn

Views are my own