Computing practices observed in industry

Jim Pivarski

Princeton University – DIANA

March 7, 2016

Context

The commercial "Big Data" movement developed more or less independently of high energy physics, even though some of the same problems had to be solved.

This talk is about the differences I have seen between the two communities, with an emphasis on technical choices, to aid in integration and interoperablity with ROOT.

Language choice

Short answer

Commercial distributed systems are usually implemented in Java, and so distributed data processing systems like Hadoop and Spark are also on the Java Virtual Machine (JVM).

Long answer

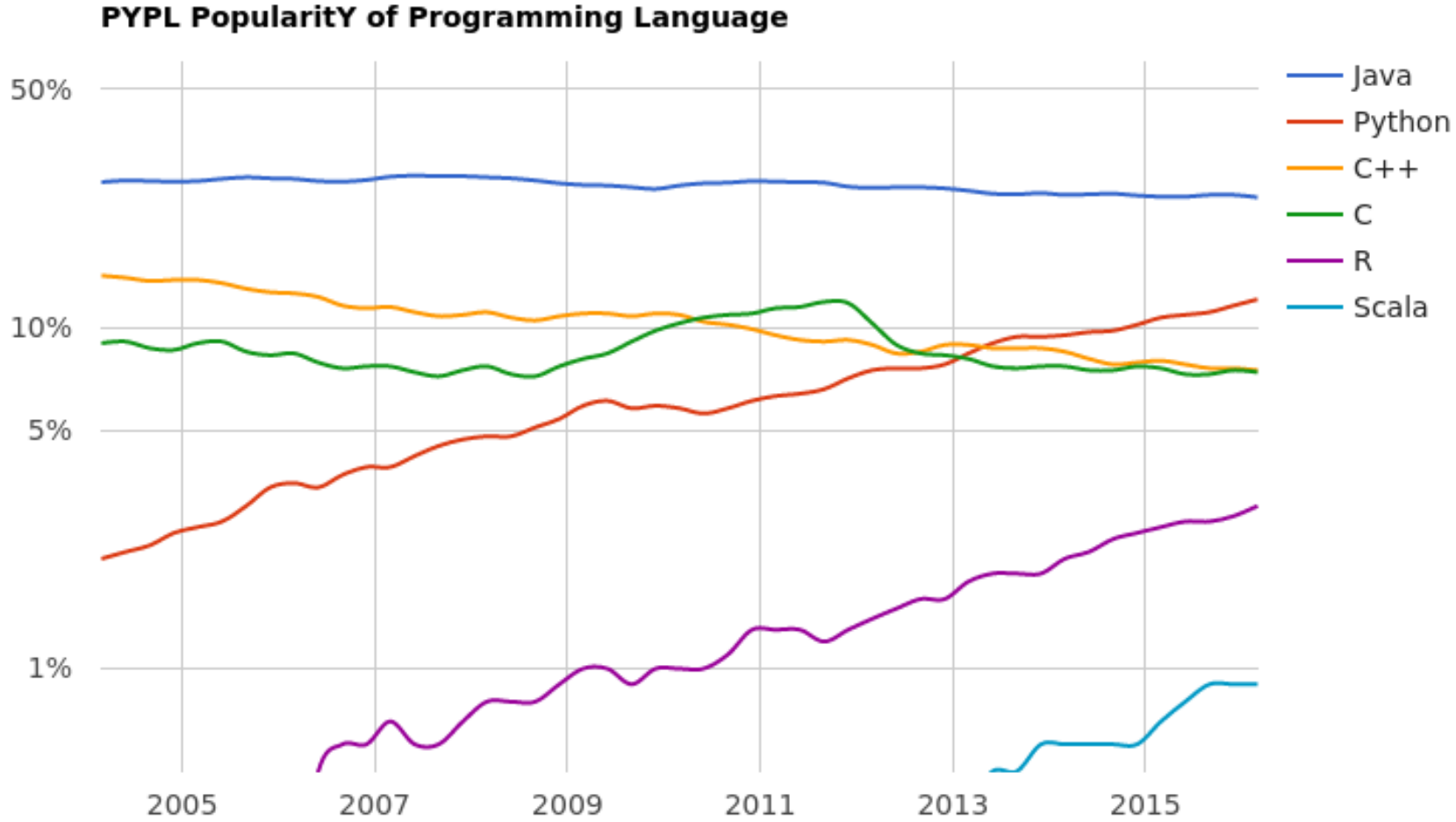
This is the topic I get the most questions about, so I'll break it down by project (next page).

Language choices for projects I was involved in

- Credit card company: SAS and Python (*pure* Python, not even Numpy).
- Web advertising start-up: Python.
- NASA (open source Project Matsu): Java simply because it was a new project using Hadoop and HBase. I think they ordinarily use C++ for image processing.
- Monitoring auto traffic: real-time analysis in Storm (which is in Clojure, a JVM language, but I wrote my code in Scala).
- Auto insurance: SQL over Hadoop, using Hive and Pig. User-defined functions were in Java because Hive and Pig (and Hadoop) are Java.
- Military project: extremely Java-centric.
- Data science start-up: most data analyses in R, a little in Python, but the production data pipeline was strictly Java.

Language choice

Based on Google searches for language + “tutorial” (log scale).



Major frameworks

Apache Hadoop

Performs map-reduce calculations. Used as a foundation for other big data frameworks because of...

- the HDFS distributed filesystem (even variants like MapR, which don't use HDFS, provide an HDFS API),
- suite of InputFormats that split files by logical records,
- ZooKeeper, which coordinates job configuration and synchronization for any service across a cluster.

Apache Spark

Generalizes from map-reduce to arbitrary pipelines, optimized for iterative procedures, with an interactive prompt. May be used on any cluster manager, but usually Hadoop.

- User interfaces: native Scala, Java, Python (through sockets), and R (through pipes).

Major frameworks

Google Trends result

(frequency of use as a search term, in "software" context)

Data pipelines and databases (frequently encountered)

It's not uncommon to use several frameworks in a single project, whether they're orthogonal in purpose or not.

Apache Storm real-time analysis, a fault-tolerant data pipeline.

Apache Drill rapid response to queries (which I think would be ideal for plotting).

Apache HBase random-access tabular database over Hadoop.

Apache Hive SQL over Hadoop (*not* random-access).

Apache Pig custom language (Pig Latin), "eats any data format."

Apache Mesos cluster manager (using ZooKeeper).

Apache Kafka message queue.

Apache Flume queues for log files.

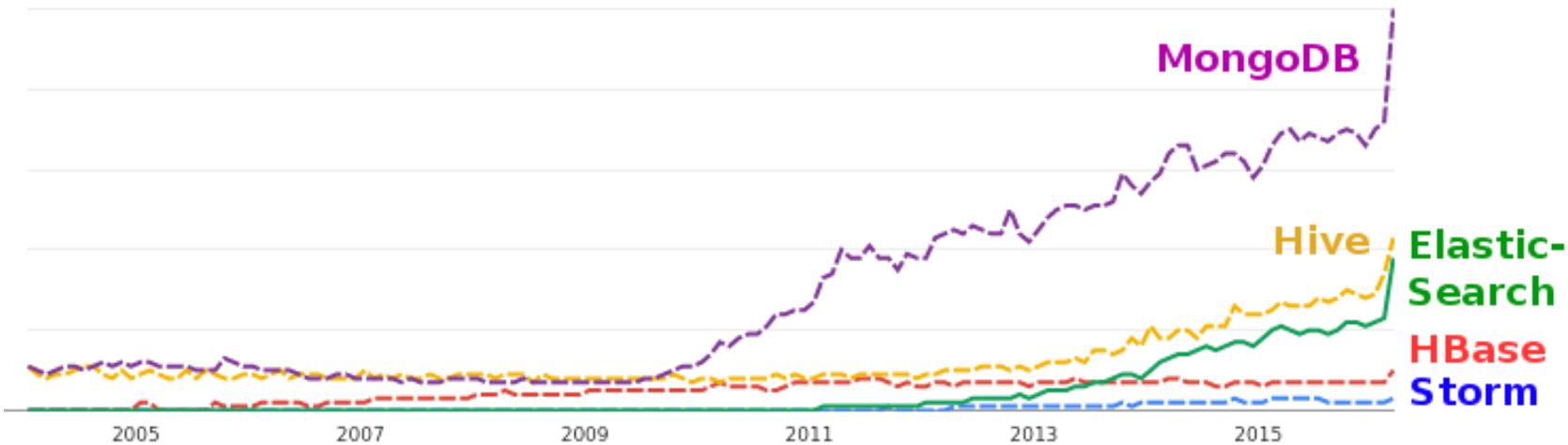
ElasticSearch full-text search engine.

MongoDB indexable JSON document store.

Major frameworks

Google Trends result

(same approximate vertical scale as before)



Generic data formats

It's also not uncommon to mix file formats and use a lot of text-based formats. Perhaps 70% of what I saw was JSON.

- **CSV** table of primitives: numbers, booleans, strings (text).
- **JSON** arrays and maps of primitives (text).
- **XML** structures with an optional schema (text).
- **Apache Avro** JSON-like binary format with algebraic data types (arrays, maps, records, and unions of primitives). Similar to Thrift and Protocol buffers.
- **Parquet** similar to Avro, but stored column-wise for speed. Similar to ORC, Hive's format for SQL tables.
- **Sequence files** structured container of arbitrary binary blobs intended as splitting hints for Hadoop.
- **Intermediate serialization in Hadoop and Spark** using native Java serialization, Kryo, and Hadoop Writables.
- **Python pickle files** for persisting Python objects.

. . . and many application-specific formats.

Machine learning

Mostly R packages, but also Mahout (over Hadoop) and MLLib (over Spark), and the Python numerical stack: Numpy, SciPy, and SciKit-Learn.

User interfaces

IPython Notebooks, RStudio, writing text files by hand (vi).

Often the only way to access the client's Hadoop cluster was through a VPN, over a Citrix Receiver, to a Windows VM with Putty installed, to a Linux head node ("ship in a bottle").

This discouraged the use of GUIs.

Some clusters were set up with Hue, to upload job JARs through a website, but the website was only accessible through these VMs.

Goals and code organization

Factorization

Hadoop is both a cluster and an analysis framework, like GRID and ROOT combined. Newer projects tend to be better factorized.

I never saw any interest in the GRID concept of sending jobs "anywhere." Even cloud computing like Amazon Web Services target specific sites, like `us-east-1`, `us-west-2`, `eu-west-1`...

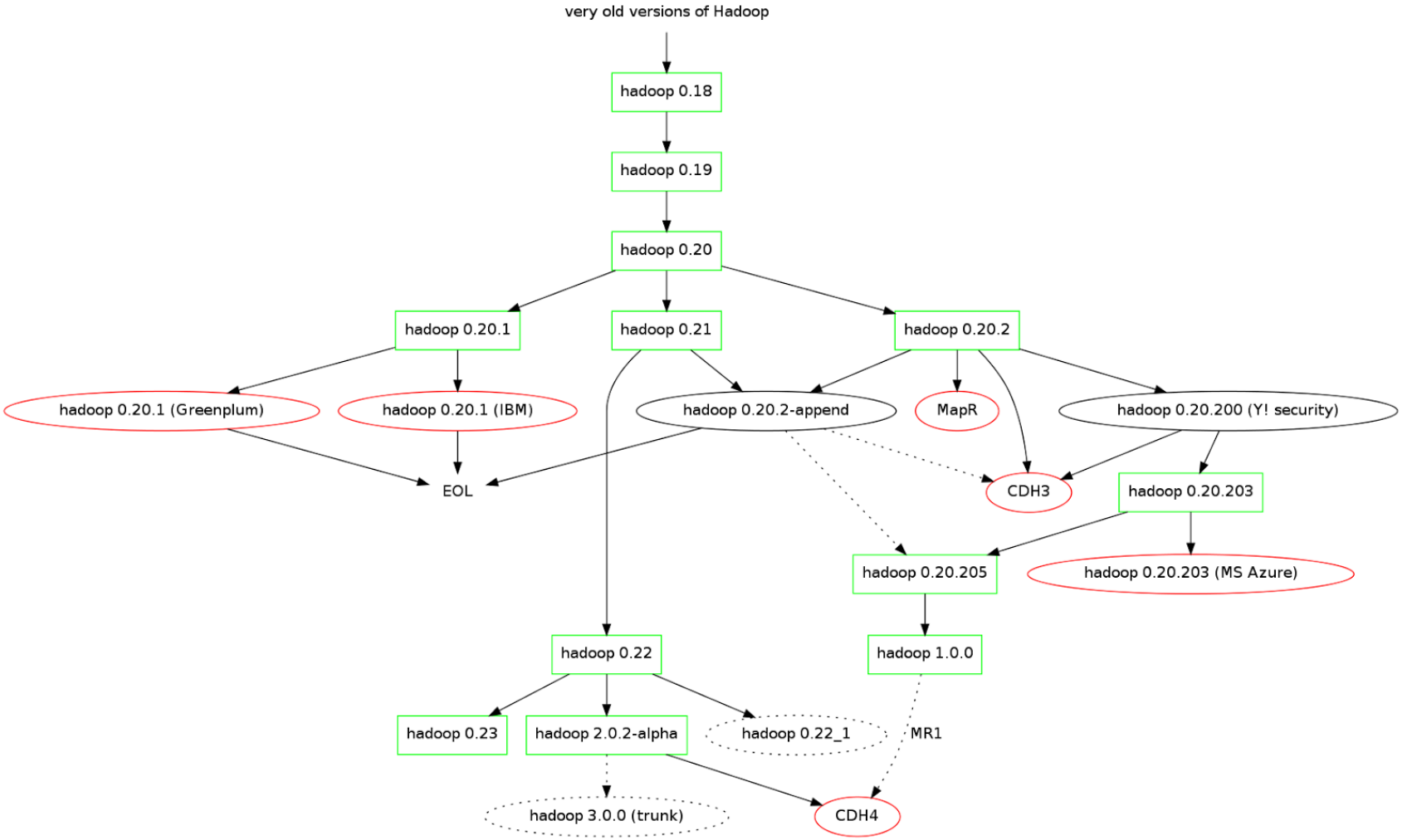
Non-centralized development

Hadoop itself is a mess:

- user code must be compiled against a Hadoop version,
- the version history is complex (0.23 is newer than 1.0),
- common features like secondary sorting are actually just common user hacks...

Pretty quickly, everyone was writing layers on top of Hadoop. Just the Python ones: hadoopy, pydoop, mrjob, dumbo, luigi, happy, hipy, I wrote one...

Helpful guide to Hadoop version control



Apache

Software projects must be cleaned up to be included in Apache (Hadoop branching resulted from competition between Apache clean-up and Cloudera/MapR/IBM/Microsoft feature requests).

The Apache Software Foundation...

- is a non-profit organization hosting open-source code.
- has business-friendly licensing (no "copyleft").
- owns and copyrights all complete projects *but also* contributors retain ownership of their individual contributions.
- presents license as a unified header block (see `http://www.apache.org/legal/src-headers.html`).
- includes competing projects that reproduce each other's functionality: Storm, Spark-Streaming, S4, Samza, Flink are all distributed stream processors.

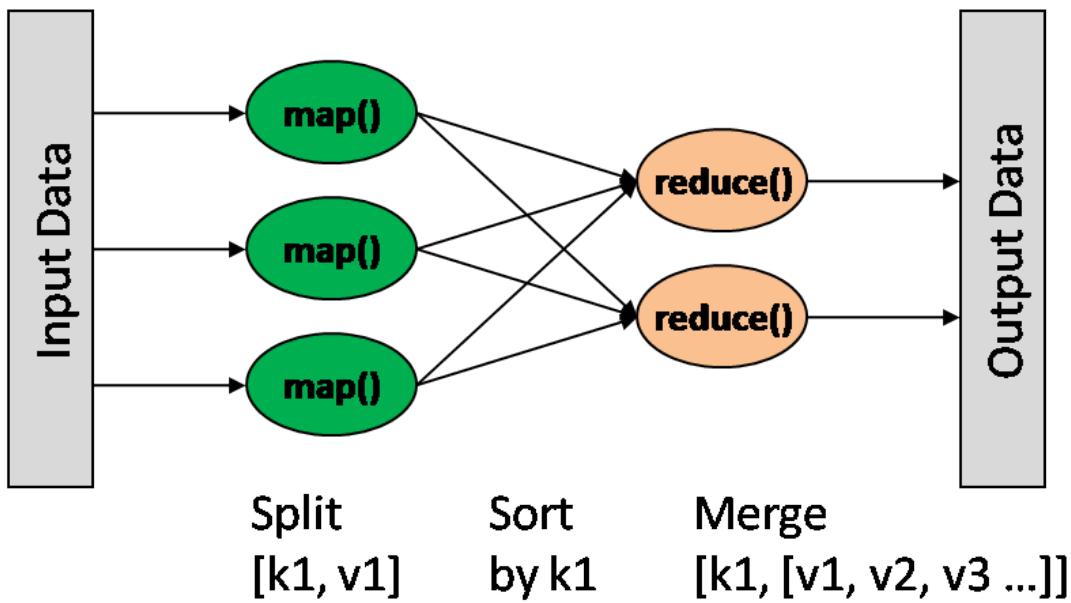
Map-reduce paradigm

User supplies a "map" function and a "reduce" function.

map converts each input `X` into key/value `pair<K,V>`.

reduce sees all values `iterator<V>` associated with a given key `K` and produces some output `Z`.

secondary sort a trick to sort `iterator<V>` as a by-product of Hadoop's key shuffling.



Spark's functional primitives in C++ type syntax

```
iterator<X> filter(iterator<X>, function<bool(X)>);
iterator<Y> map(iterator<X>, function<Y(X)>);
iterator<Y> flatten(iterator<iterator<Y>>);
iterator<Y> flatMap(iterator<X>, function<iterator<Y>(X)>);

Y reduce(iterator<Y>, function<Y(Y,Y)>);       // sum, max, etc.
Z aggregate(iterator<Y>, Z, function<Z(Y,Z)>);   // more general

map<K,array<V>> groupByKey(iterator<pair<K,V>>); // like SQL's

map<K,V> reduceByKey(iterator<pair<K,V>>,      // input data
                     function<V(V,V)>);        // merge

map<K,Z> aggregateByKey(iterator<pair<K,V>>,   // input data
                        Z,                     // starting value
                        function<Z(V,Z)>,      // increment
                        function<Z(Z,Z)>);     // combine
```

Hadoop's "mappers" are actually `flatMap` (which includes the possibility of filtering) and its "reducers" are `aggregateByKey`.

Spark's functional primitives in Scala type syntax

```
def filter(in: Iterator[X], f: X => Boolean): Iterator[X]
def map(in: Iterator[X], f: X => Y): Iterator[Y]
def flatten(in: Iterator[Iterator[Y]]): Iterator[Y]
def flatMap(in: Iterator[X], f: X => Iterator[Y]): Iterator[Y]

def reduce(in: Iterator[Y], f: (Y,Y) => Y): Y
def aggregate(in: Iterator[Y], zero: Z, f: (Y,Z) => Z): Z

def groupByKey(in: Iterator[(K,V)]): Map[K,Seq[V]]

def reduceByKey(in: Iterator[(K,V)],
                merge: (V,V) => V): Map[K,V]

def aggregateByKey(in: Iterator[(K,V)],
                   zero: Z,
                   incr: (V,Z) => Z,
                   comb: (Z,Z) => Z): Map[K,Z]
```

Hadoop's "mappers" are actually `flatMap` (which includes the possibility of filtering) and its "reducers" are `aggregateByKey`.

Example of using primitives

Suppose we want to fill histograms of p_T for tracks with `fValid` from events with `fTemperature > 20`, with one histogram for each distinct combination of `fTriggerBits`.

On the Spark prompt (Scala syntax):

```
val dataset = sc.rootRDD[Event]("root://fnal.gov/*.root")
val histograms =
  dataset.filter(event => event.fTemperature > 20)
         .flatMap(event => event.fTracks)
         .filter(track => track.fValid)
         .map(t => (t.fTriggerBits,
                    Math.sqrt(t.fPx*t.fPx + t.fPy*t.fPy)))
         .aggregateByKey(new TH1F(100, 0, 30))(
                         (pt, hist) => hist.Fill(pt),
                         (h1, h2) => h1.Add(h2))
```

This process gets distributed over the cluster and returns its result to the user's Spark interactive session.

Example of using primitives

Suppose we want to fill histograms of p_T for tracks with `fValid` from events with `fTemperature > 20`, with one histogram for each distinct combination of `fTriggerBits`.

On the PySpark prompt (Python syntax):

```
dataset = sc.rootRDD("root://fnal.gov/*.root")
histograms =
  dataset.filter(lambda event: event.fTemperature > 20)
         .flatMap(lambda event: event.fTracks)
         .filter(lambda track: track.fValid)
         .map(lambda t: (t.fTriggerBits,
                         math.sqrt(t.fPx**2 + t.fPy**2)))
         .aggregateByKey(TH1F(100, 0, 30),
                         lambda pt, hist: hist.Fill(pt),
                         lambda h1, h2: h1.Add(h2))
```

This process gets distributed over the cluster and returns its result to the user's PySpark interactive session.