Distributed Computing with Spark

02807 Computational Tools for Data Science

Plan for rest of the course

2019-11-12: Introduction to Spark

2019-11-19: Spark under the hood

2019-11-26: Guest lecture and project work

2019-12-03: Guest lecture and project work

2019-12-05: Hand-in project #3 and mandatory assignment #3

Talk Announcement

Title: **Working with Data at Warp Speed**

Speaker: **Brian Bell** (datarobot.com)

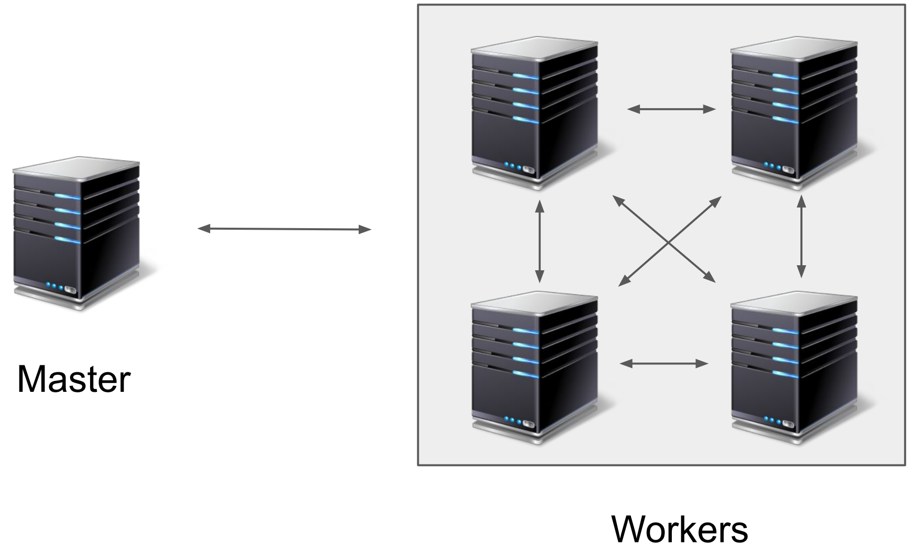
Date: **2019-11-26**

Today

- Distributed computing
- Spark
  - Spark dataframes
  - PySpark
  - Spark UI

Distributed Computing

Computation is distributed among multiple computers (nodes)



Distributed Computing

Pros:

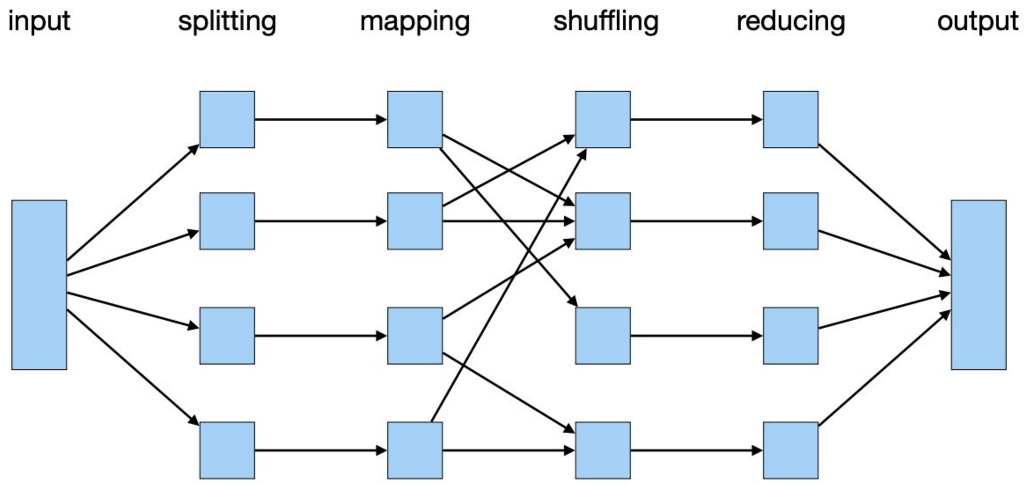
- Speeding up computation
- Spreading risk of crashing

Cons:

- Exchanging data between workers is expensive (bottleneck: network)
- Synchronization is expensive (bottleneck: disk)
- Writing distributed algorithms is hard

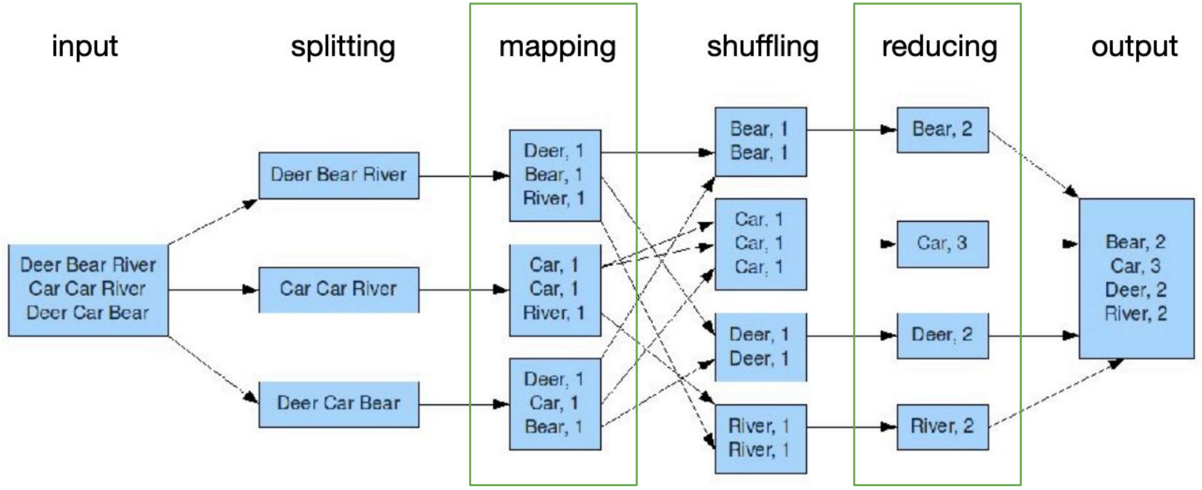
Distributed Computing

- Writing distributed algorithms is hard, but...
- When processing data, the MapReduce model makes it simple



MapReduce model example

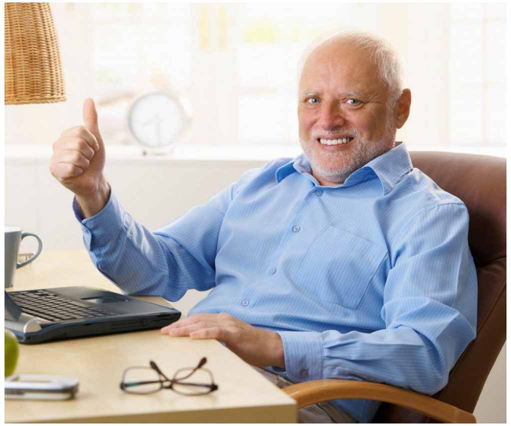
- Input: Document of words (Deer Bear River Car Car River Deer Car Bear)
- Output: The frequency of each word



- Frameworks like Spark enables you to provide map and reduce functions - everything else is taken care of by the framework

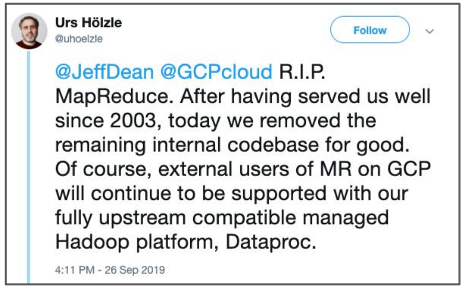
But I don’t have cluster!

- Modern laptops are single node, multi-core systems
- We want full utilization of our local resources
- Programs written in Spark will run locally as well as on clusters without modification



Introducing Apache Spark

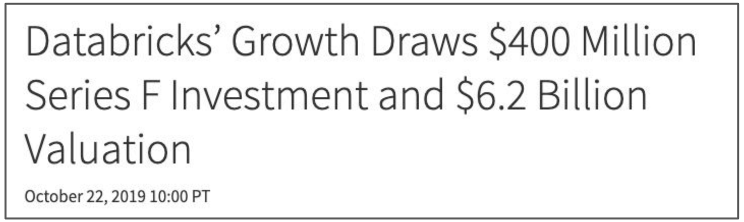
- Distributed computing framework
- Invented at Berkeley
- Written in Scala
- Free
- Has a growing community
- Becoming de-facto standard in the industry
- Faster than its predecessors



Urs Hölzle
@uhoelzle
Follow

@JeffDean @GCPcloud R.I.P. MapReduce. After having served us well since 2003, today we removed the remaining internal codebase for good. Of course, external users of MR on GCP will continue to be supported with our fully upstream compatible managed Hadoop platform, Dataproc.

4:11 PM - 26 Sep 2019



Databricks' Growth Draws $400 Million Series F Investment and $6.2 Billion Valuation

October 22, 2019 10:00 PT

Spark feature highlights

- Built-in resilience
- Node awareness
- Supports many file formats/data sources
- Streaming
- Same code, many platforms
- High-level API

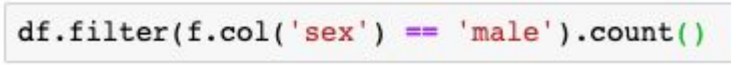
Spark DataFrame

- High level/declarative SQL-like API
- Immutable and lazy evaluation of commands
- Automatic optimization
- No random access (direct access to a specific row)

Spark DataFrame: High level/declarative SQL-like API

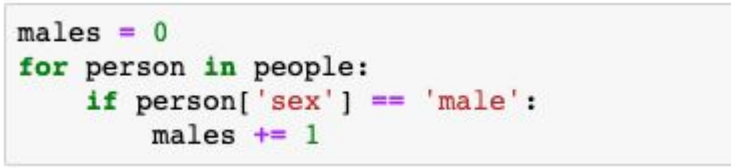
- You tell Spark *what* to do, not *how* to do it
- Example: counting males

What:



```
df.filter(f.col('sex') == 'male').count()
```

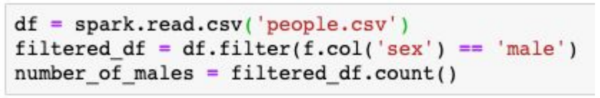
How:



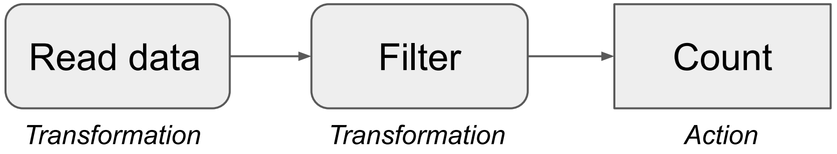
```
males = 0
for person in people:
    if person['sex'] == 'male':
        males += 1
```

Spark DataFrame: Immutability and lazy evaluation

- Immutability: Every operation produces a new dataframe
- Two kinds of operations: transformations and actions
  - A transformation creates a new dataframe, but nothing is actually computed (*lazy* operation)
  - An action triggers an actual computation
- Example:



```
df = spark.read.csv('people.csv')
filtered_df = df.filter(f.col('sex') == 'male')
number_of_males = filtered_df.count()
```



Spark DataFrame: Automatic optimization

- Spark re-organizes your operations to optimize performance
- More about this next week!

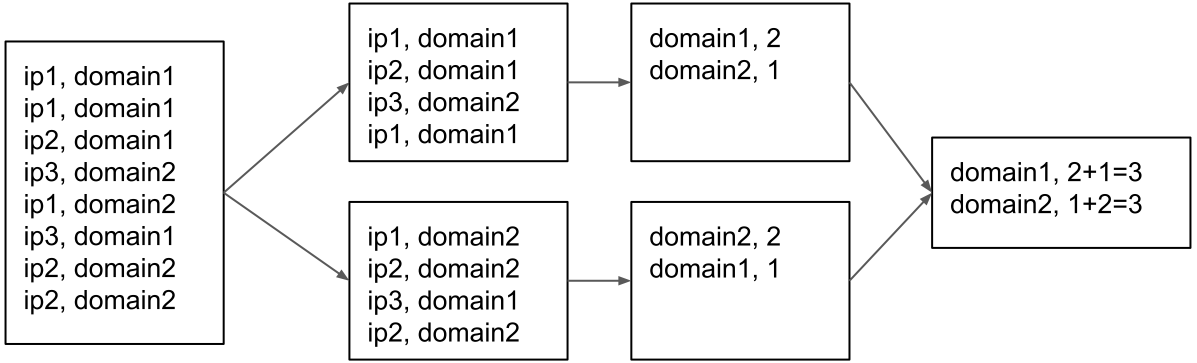
Spark DataFrame: No random access

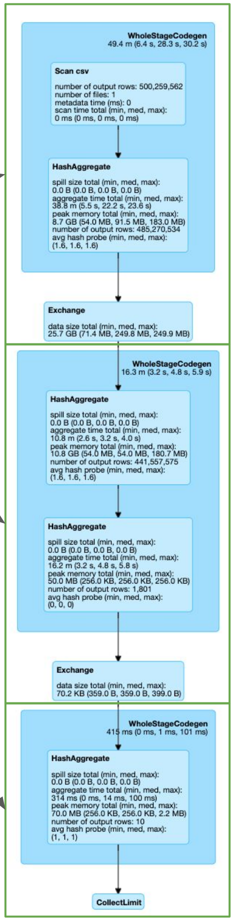
- You cannot access row *i* in a dataframe the same was in a Python list
- After the split, the master knows the size of the split, but doesn’t know which rows that went into which splits
- You **have to** think in terms of the MapReduce model to leverage Spark

Let’s code

Job analysis

- Compute hash(ip, domain) for each row and use value to decide splitting
- For each split, compute uniques per domain
- Compute hash(domain) to group domains in new splits
- Sum up the uniques





```
WholeStageCodegen
49.4 m (6.4 s, 28.3 s, 30.2 s)

Scan csv
number of output rows: 500,259,562
number of files: 1
metadata time (ms): 0
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)

HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
38.8 m (5.5 s, 22.2 s, 23.6 s)
peak memory total (min, med, max):
8.7 GB (54.0 MB, 91.5 MB, 183.0 MB)
number of output rows: 485,270,534
avg hash probe (min, med, max):
(1.6, 1.6, 1.6)

Exchange
data size total (min, med, max):
25.7 GB (71.4 MB, 249.8 MB, 249.9 MB)

WholeStageCodegen
16.3 m (3.2 s, 4.8 s, 5.9 s)

HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
10.8 m (2.6 s, 3.2 s, 4.0 s)
peak memory total (min, med, max):
10.8 GB (54.0 MB, 54.0 MB, 180.7 MB)
number of output rows: 441,557,575
avg hash probe (min, med, max):
(1.6, 1.6, 1.6)

HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
16.2 m (3.2 s, 4.8 s, 5.8 s)
peak memory total (min, med, max):
50.0 MB (256.0 KB, 256.0 KB, 256.0 KB)
number of output rows: 1,801
avg hash probe (min, med, max):
(0, 0, 0)

Exchange
data size total (min, med, max):
70.2 KB (359.0 B, 359.0 B, 399.0 B)

WholeStageCodegen
415 ms (0 ms, 1 ms, 101 ms)

HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
314 ms (0 ms, 14 ms, 100 ms)
peak memory total (min, med, max):
70.0 MB (256.0 KB, 256.0 KB, 2.2 MB)
number of output rows: 10
avg hash probe (min, med, max):
(1, 1, 1)

CollectLimit
```