Big data systems

12/8/17

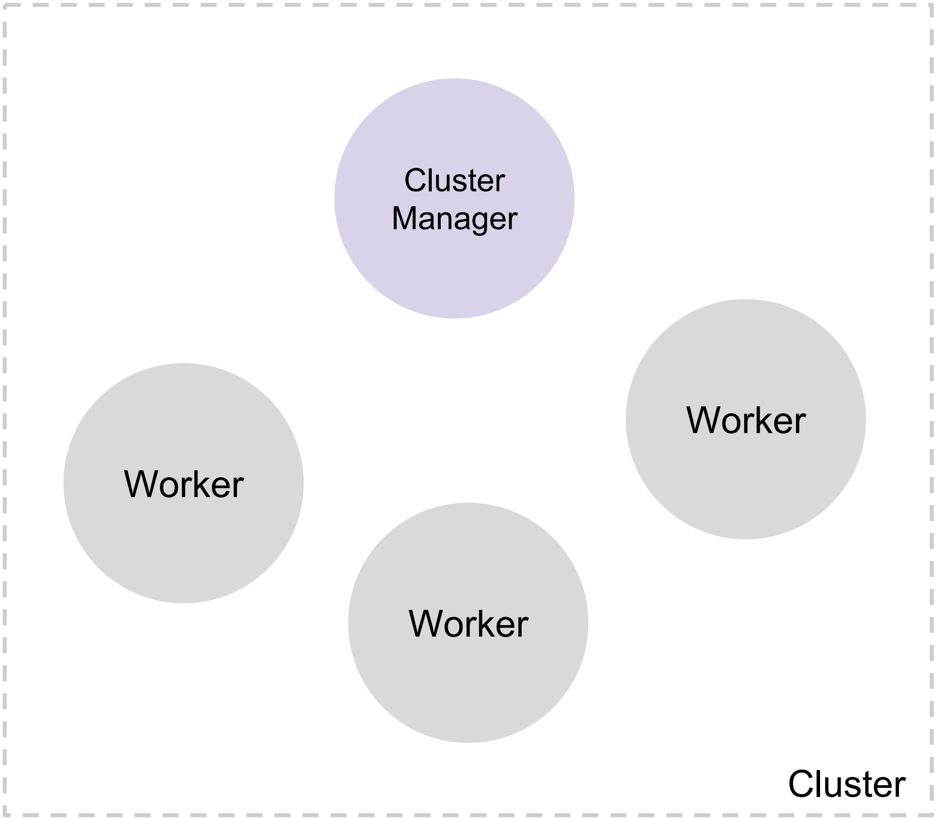
Today

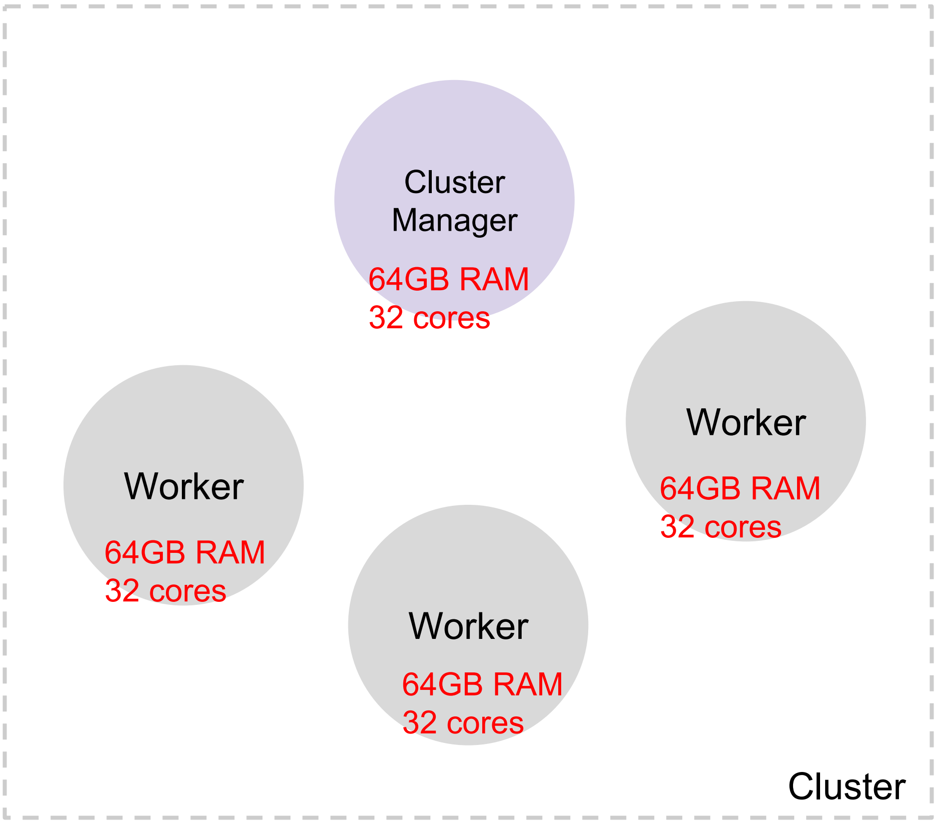
Basic architecture

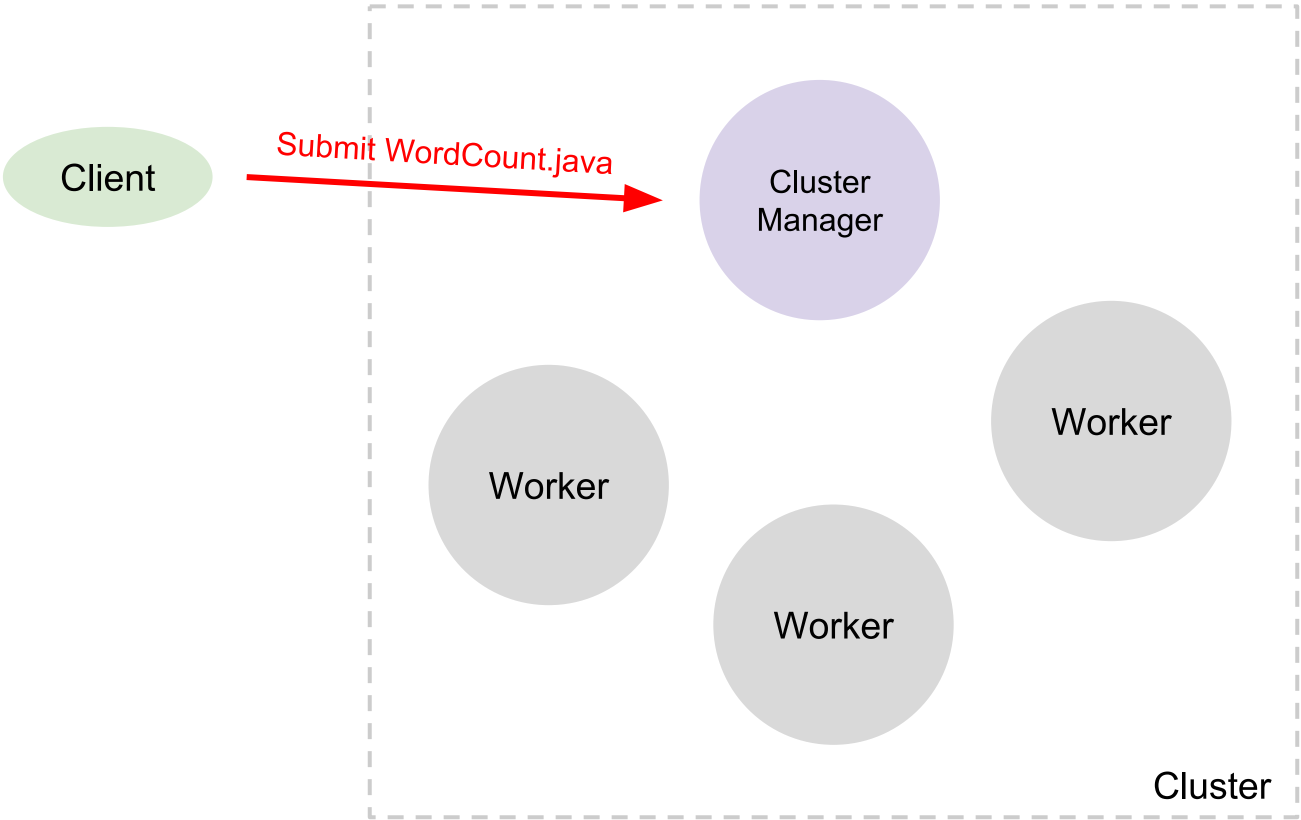
Two levels of scheduling

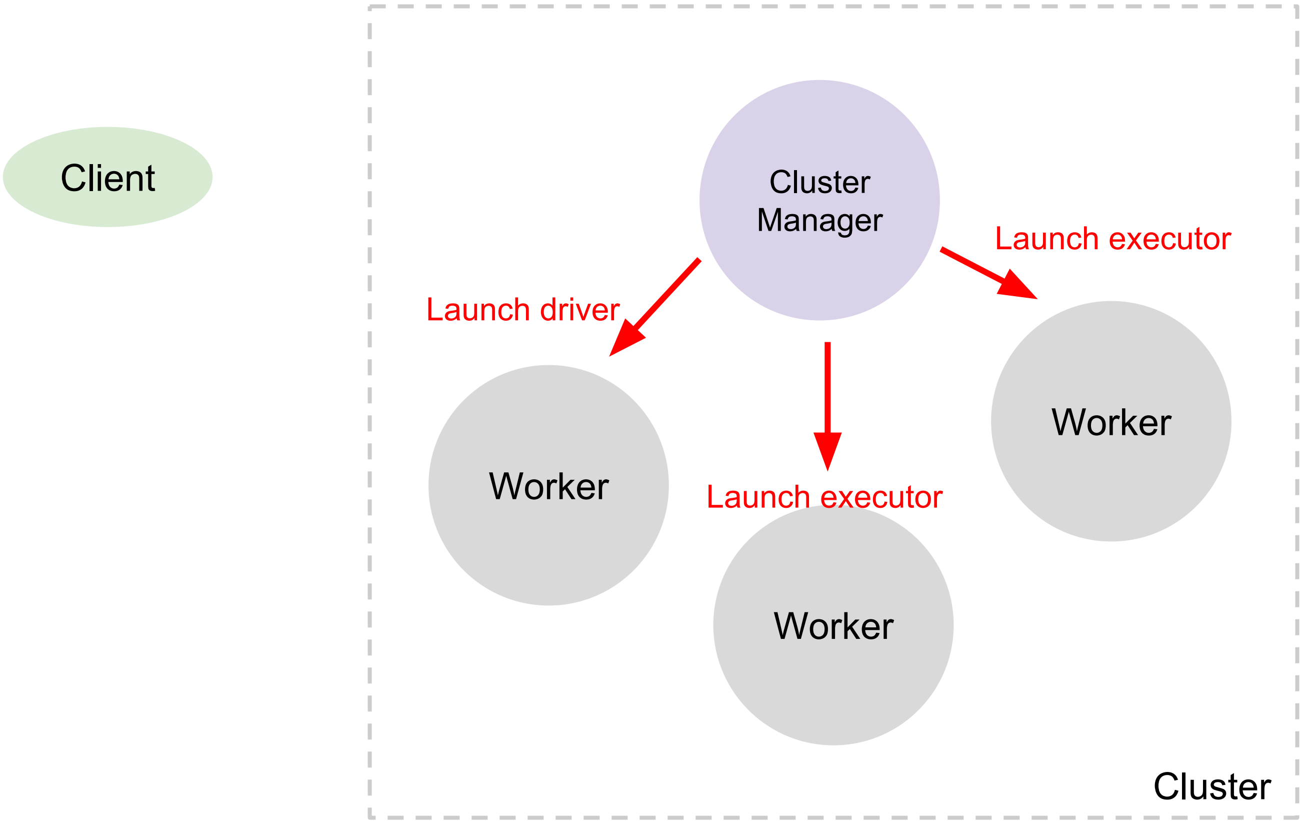
Spark overview

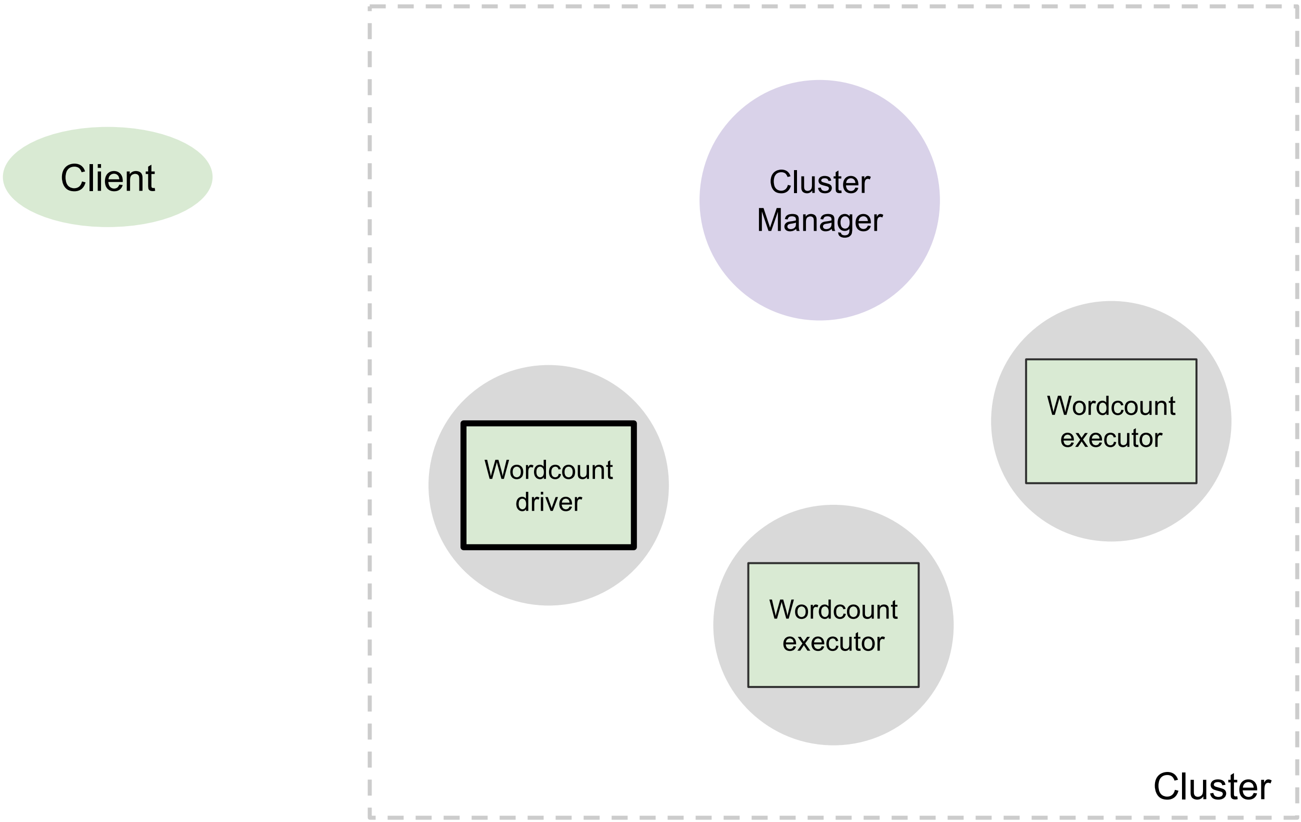
Basic architecture

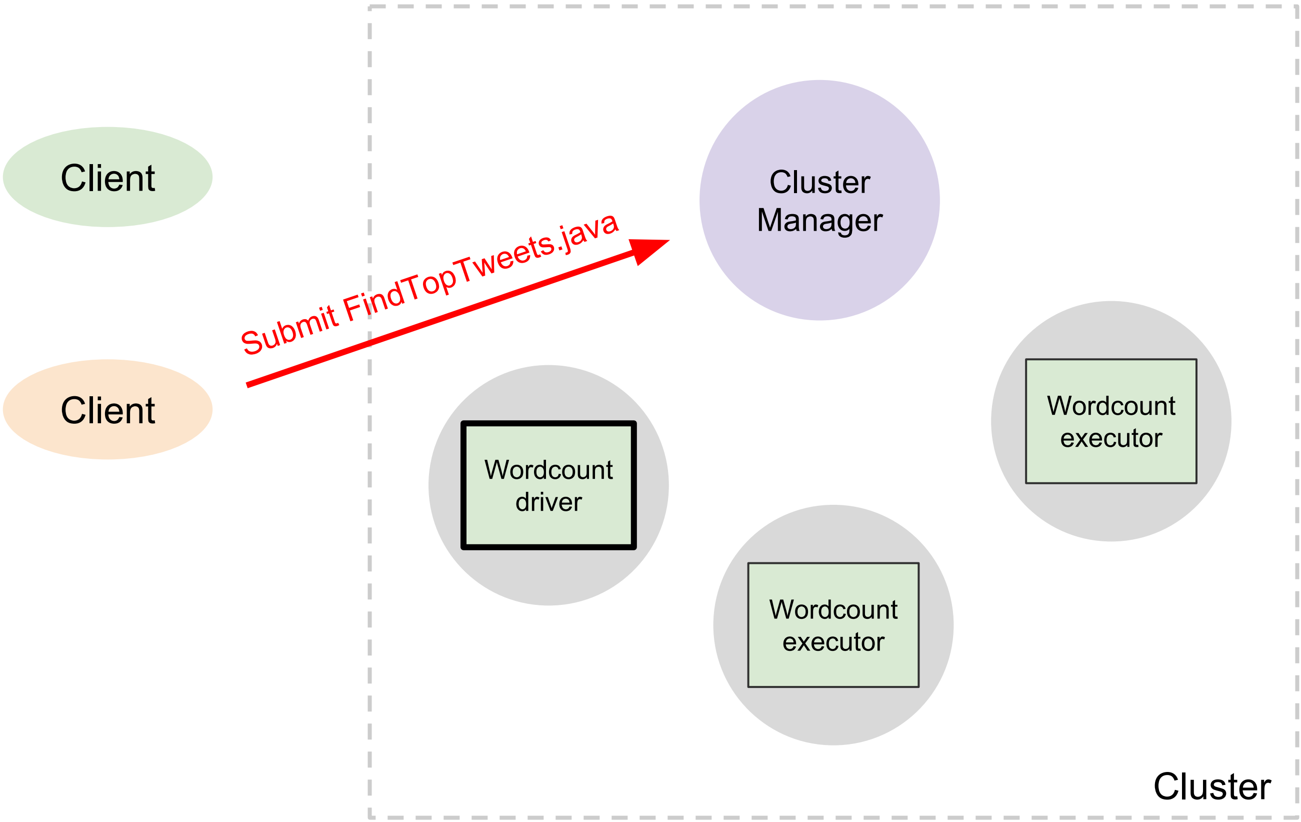

Cluster
Manager
Worker
Worker
Worker
Cluster

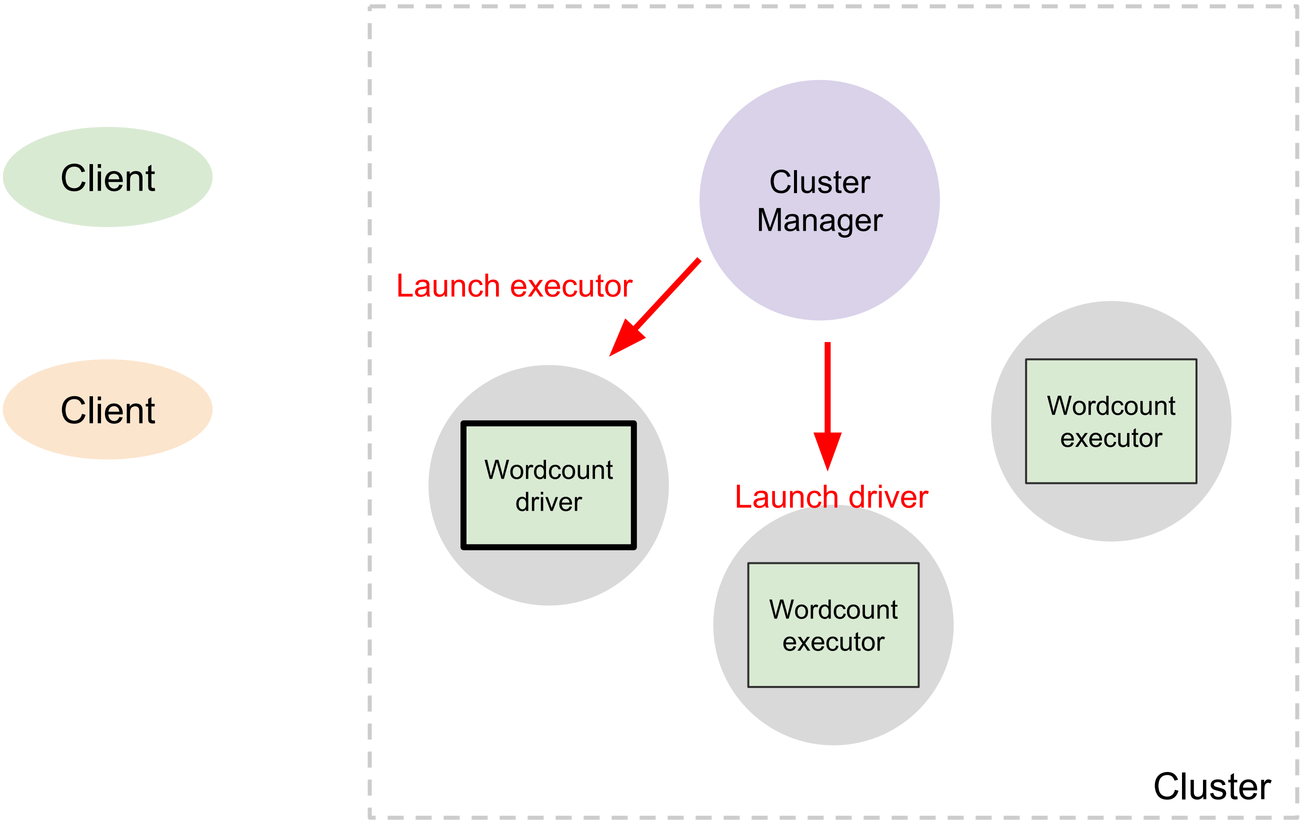

Cluster Manager
64GB RAM
32 cores
Worker
64GB RAM
32 cores
Worker
64GB RAM
32 cores
Worker
64GB RAM
32 cores
Cluster

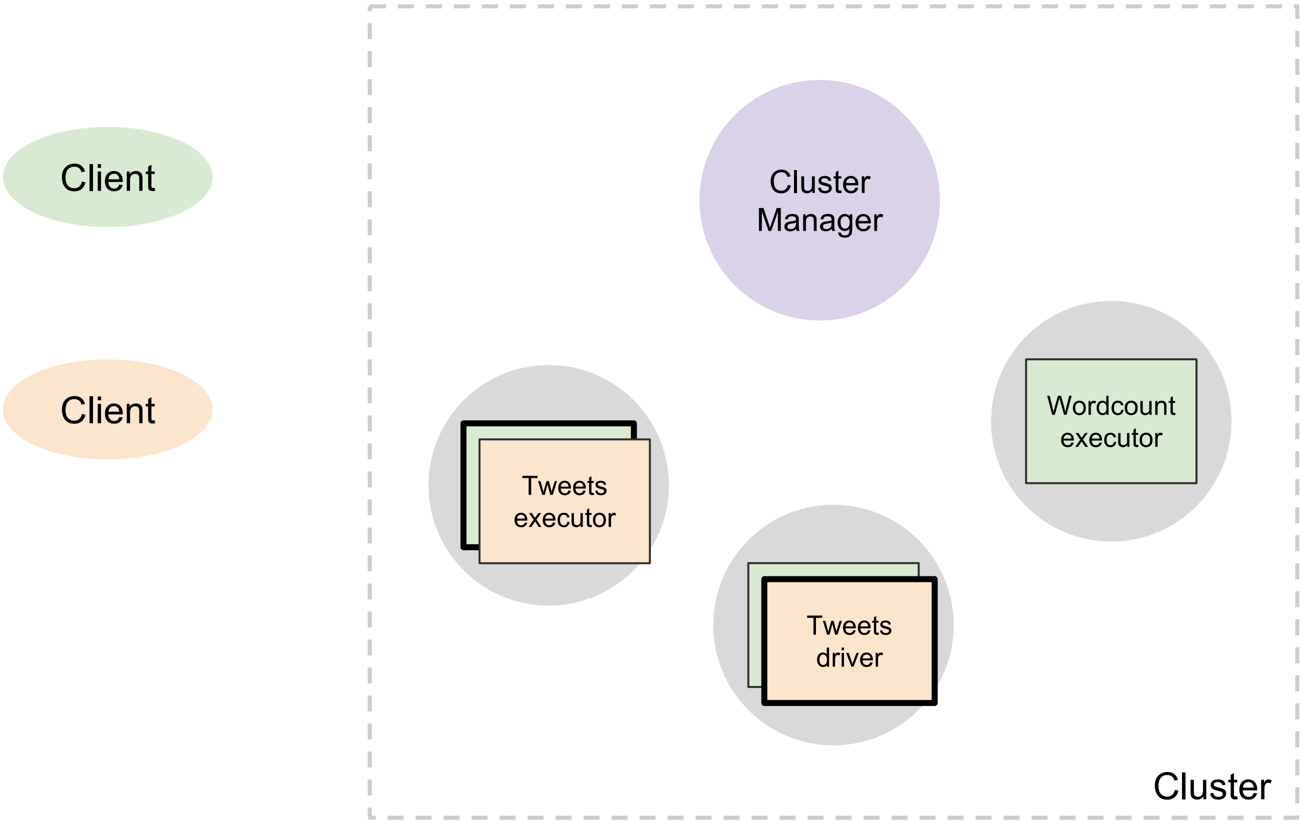

Client
Submit WordCount.java
Cluster Manager
Worker
Worker
Worker
Cluster

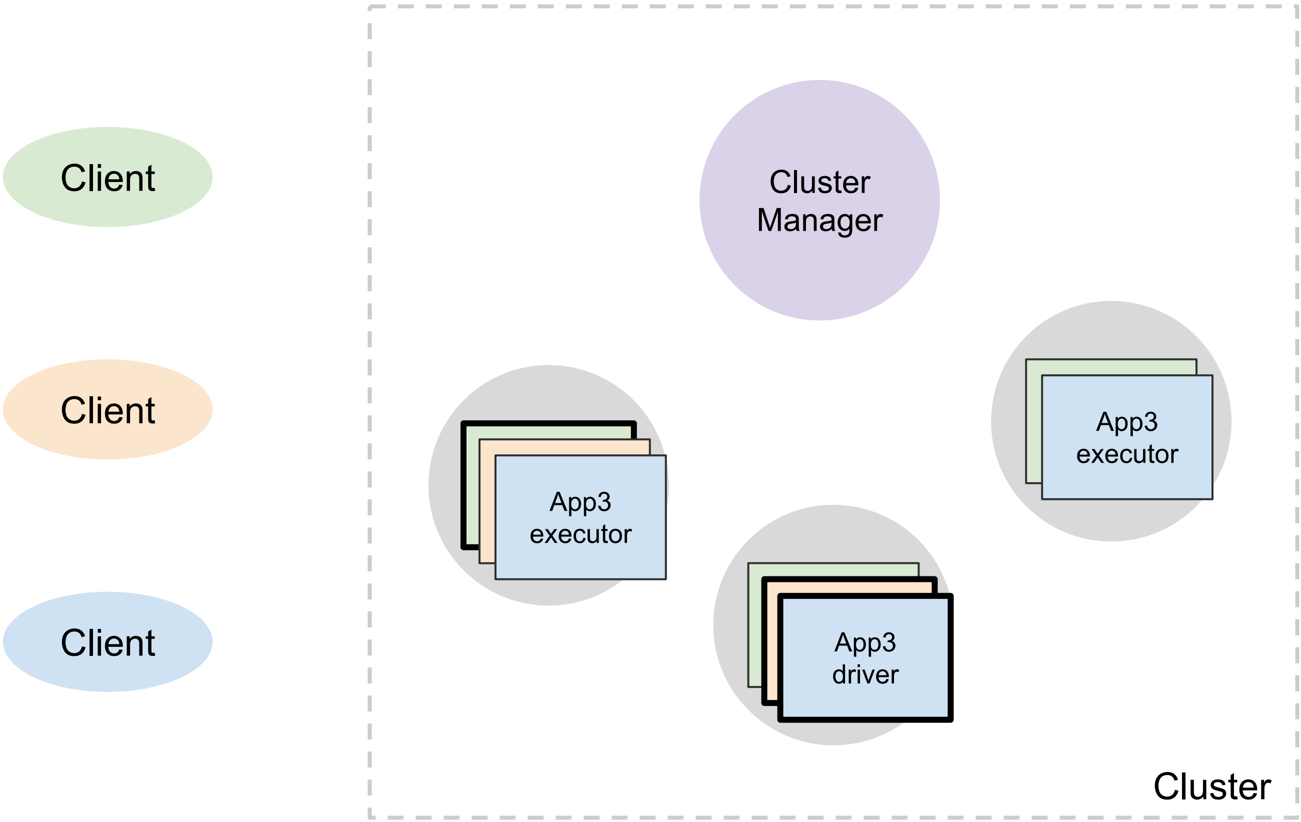

Client
Cluster Manager
Launch driver
Launch executor
Launch executor
Worker
Worker
Worker
Cluster


Client
Cluster Manager
Wordcount driver
Wordcount executor
Wordcount executor
Cluster


Client
Client
Submit FindTopTweets.java
Cluster
Manager
Wordcount
driver
Wordcount
executor
Wordcount
executor
Cluster


Client
Client
Cluster Manager
Launch executor
Launch driver
Wordcount driver
Wordcount executor
Wordcount executor
Cluster


Client
Client
Cluster
Manager
Tweets
executor
Wordcount
executor
Tweets
driver
Cluster


Client
Client
Client
Cluster
Manager
App3
executor
App3
executor
App3
driver
Cluster

Basic architecture

Clients submit applications to the cluster manager

Cluster manager assigns cluster resources to applications

Each Worker launches containers for each application

Driver containers run main method of user program

Executor containers run actual computation

Two levels of scheduling

Cluster-level: Cluster manager assigns resources to applications

Application-level: Driver assigns *tasks* to run on executors

 A task is a unit of execution that corresponds to one *partition* of your data

E.g. Running Spark on a YARN cluster

 Hadoop YARN is a cluster scheduling framework

 Spark is a distributed computing framework

Two levels of scheduling

Cluster-level: Cluster manager assigns resources to applications

Application-level: Driver assigns *tasks* to run on executors

A task is a unit of execution that corresponds to one *partition* of your data

What are some advantages of having two levels?

Applications need not be concerned with resource fairness

Cluster manager need not be concerned with individual tasks (too many)

Easy to implement priorities and preemption

Spark overview

What is Apache Spark™?

Fast and general engine for big data processing

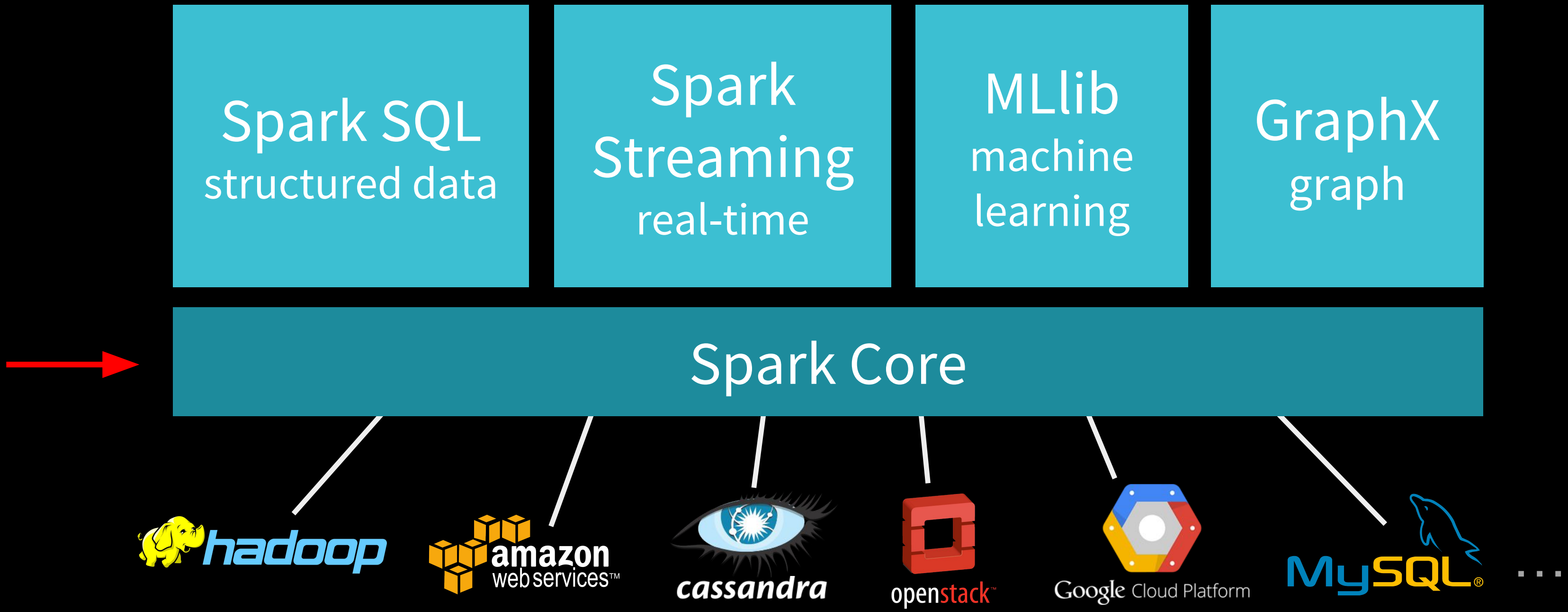
Fast to *run* code

- In-memory data sharing
- General computation graphs

Fast to *write* code

- Rich APIs in Java, Scala, Python
- Interactive shell

What is Apache Spark™?



Spark computation model

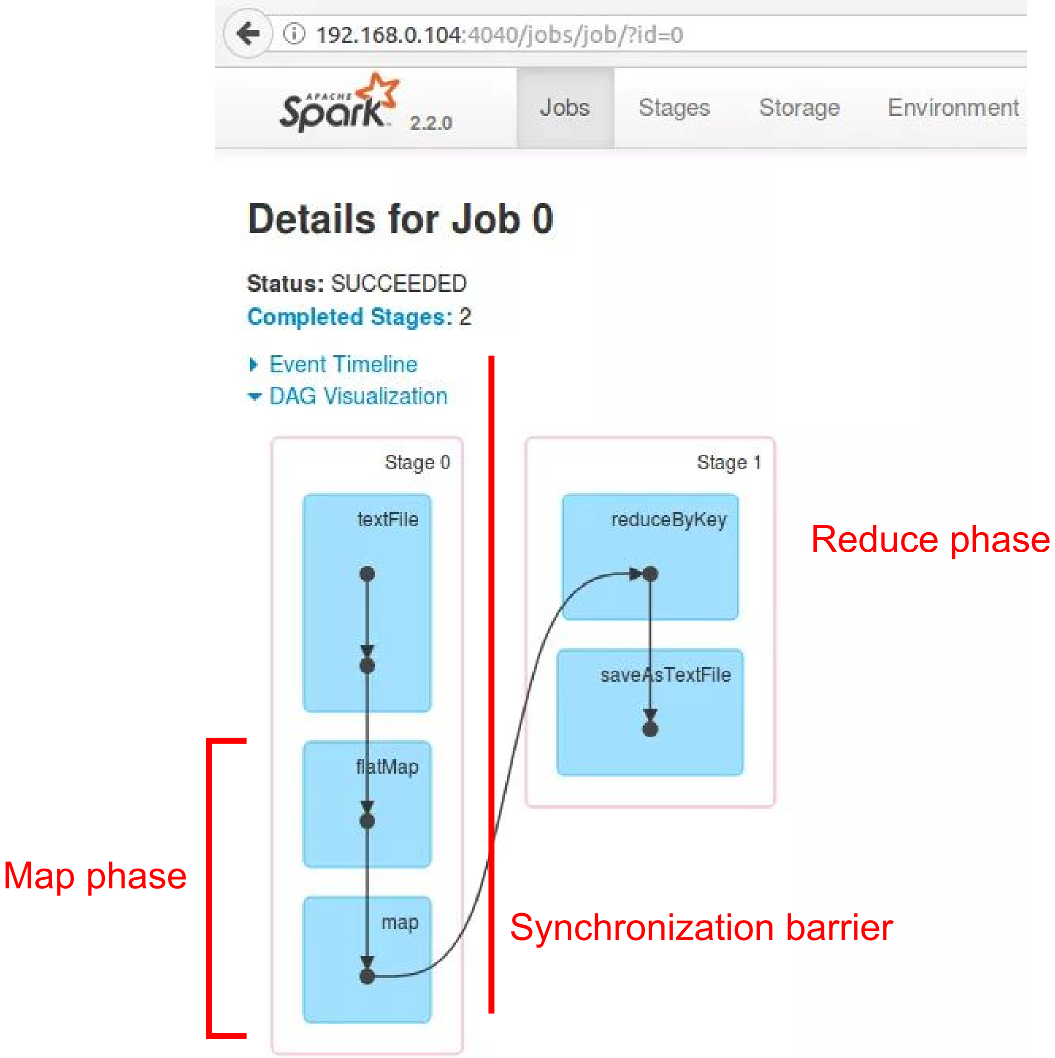
Recall that…

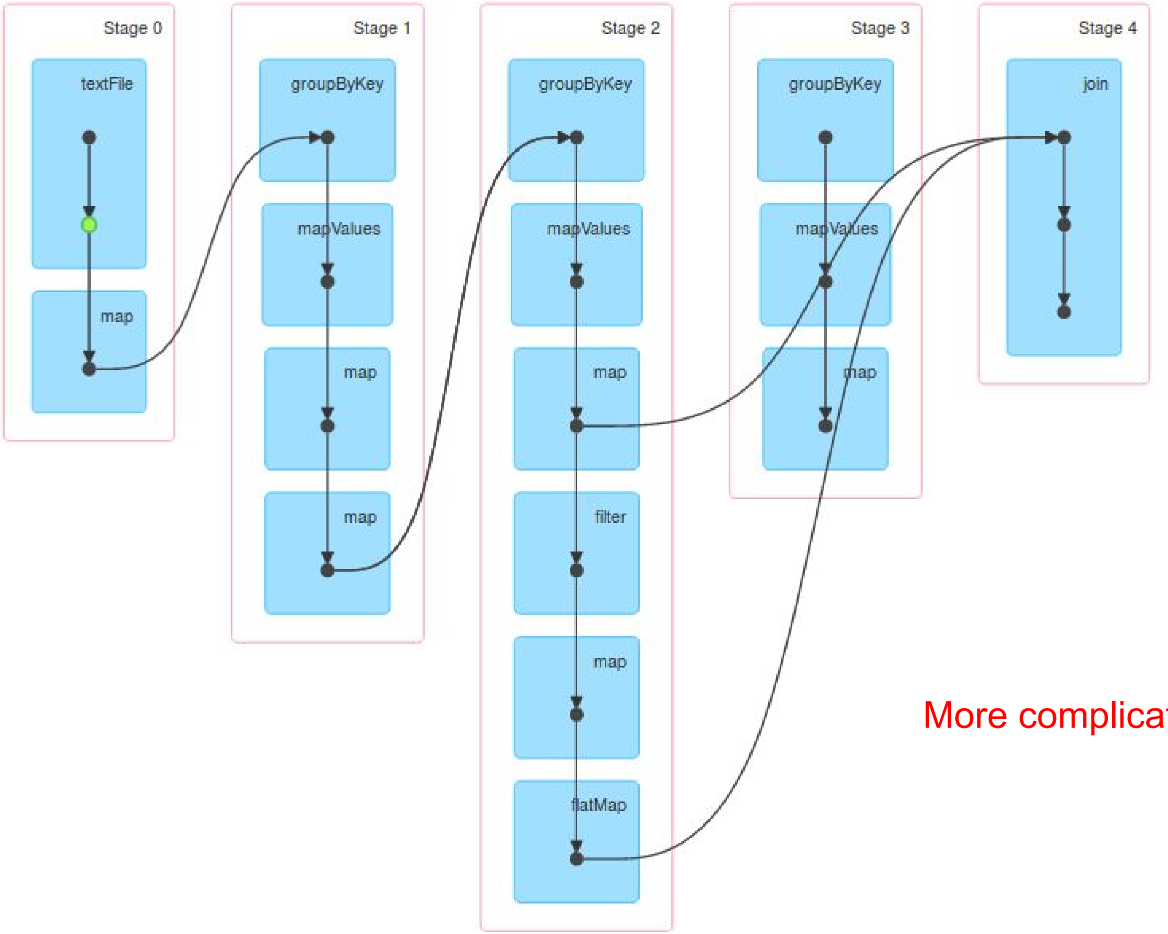
Map phase defines how each machine processes its individual partition

Reduce phase defines how to merge map outputs from previous phase

Most computation can be expressed in terms of these two phases

Spark expresses computation as a DAG of maps and reduces


192.168.0.104:4040/jobs/job/?id=0
Spark 2.2.0
Jobs
Stages
Storage
Environment
Details for Job 0
Status: SUCCEEDED
Completed Stages: 2
Event Timeline
DAG Visualization
Stage 0
textFile
flatMap
map
Stage 1
reduceByKey
saveAsTextFile
Reduce phase
Map phase
Synchronization barrier



More complicated DAG

Spark basics

Transformations express how to process a dataset

Actions express how to turn a transformed dataset into results

```
sc.textFile("declaration-of-independence.txt")
    .flatMap { line => line.split(" ") }
    .map { word => (word, 1) }
    .reduceByKey { case (counts1, counts2) => counts1 + counts2 }
    .collect()
```

Spark basics

Transformations express how to process a dataset

Actions express how to turn a transformed dataset into results

```
sc.textFile("declaration-of-independence.txt") // input data
    .flatMap { line => line.split(" ") }
    .map { word => (word, 1) }
    .reduceByKey { case (counts1, counts2) => counts1 + counts2 }
    .collect()
```

Spark basics

Transformations express how to process a dataset

Actions express how to turn a transformed dataset into results

```
sc.textFile("declaration-of-independence.txt")
    .flatMap { line => line.split(" ") } // transformations
    .map { word => (word, 1) }
    .reduceByKey { case (counts1, counts2) => counts1 + counts2 }
    .collect()
```

Spark basics

Transformations express how to process a dataset

Actions express how to turn a transformed dataset into results

```
sc.textFile("declaration-of-independence.txt")
    .flatMap { line => line.split(" ") }
    .map { word => (word, 1) }
    .reduceByKey { case (counts1, counts2) => counts1 + counts2 }
    .collect() // action
```

Spark optimization #1

Transformations can be pipelined until we hit

- A synchronization barrier (e.g. reduce), or
- An action

Example:

```
data.map { ... }.filter { ... }.flatMap { ... }.groupByKey().count()
```

These three operations can all be run in the same task

This allows lazy execution; we don't need to eagerly execute map

Spark optimization #2

In-memory caching: store intermediate results in memory to bypass disk access

Example:

```
val cached = data.map { ... }.filter { ... }.cache()
(1 to 100).foreach { i =>
    cached.reduceByKey { ... }.saveAsTextFile(...)
}
```

By caching transformed data in memory, we skip the map, the filter, and reading the original data from disk every iteration

Spark optimization #3

Reusing map outputs (aka shuffle files) allows Spark to skip map stages

Along with caching, this makes iterative workloads much faster

Example:

```
val transformed = data.map { ... }.filter { ... }.reduceByKey { ... }
transformed.collect()
transformed.collect() // does not run map phase again
```

Recap

Spark is expressive because its computation model is a DAG

Spark is fast because of many optimizations, in particular:

- Pipelined transformations
- In-memory caching
- Reusing map outputs

Brainstorm: Top K

Top K is the problem of finding the largest K values from a set of numbers

How would you express this as a distributed application?

In particular, what would the map and reduce phases look like?

Brainstorm: Top K

Top K is the problem of finding the largest K values from a set of numbers

How would you express this as a distributed application?

In particular, what would the map and reduce phases look like?

Hint: use a heap...

Top K

Assuming that a set of K integers fit in memory…

Key idea...

Map phase: everyone maintains a heap of K elements

Reduce phase: merge the heaps until you're left with one

Top K

Problem: What are the keys and values here?

No notion of key here, just assign one key to all the values (e.g. key = 1)

Map task 1: [10, 5, 3, 700, 18, 4] → (1, heap(700, 18, 10))

Map task 2: [16, 4, 523, 100, 88] → (1, heap(523, 100, 88))

Map task 3: [3, 3, 3, 3, 300, 3] → (1, heap(300, 3, 3))

Map task 4: [8, 15, 20015, 89] → (1, heap(20015, 89, 15))

Then all the heaps will go to a single reducer responsible for the key 1

This works, but clearly not scalable...

Top K

Idea: Use X different keys to balance load (e.g. X = 2 here)

Map task 1: [10, 5, 3, 700, 18, 4] → (1, heap(700, 18, 10))

Map task 2: [16, 4, 523, 100, 88] → (1, heap(523, 100, 88))

Map task 3: [3, 3, 3, 3, 300, 3] → (2, heap(300, 3, 3))

Map task 4: [8, 15, 20015, 89] → (2, heap(20015, 89, 15))

Then all the heaps will (hopefully) go to X different reducers

Rinse and repeat (*what's the runtime complexity?*)