Data-Intensive Distributed Computing

CS 431/461 451/651 (Fall 2019)

Part 2: From MapReduce to Spark (2/2)

Ali Abedi

These slides are available at http://roegiest.com/bigdata-2019w/

YARN

Hadoop’s (original) limitations:

Can only run MapReduce

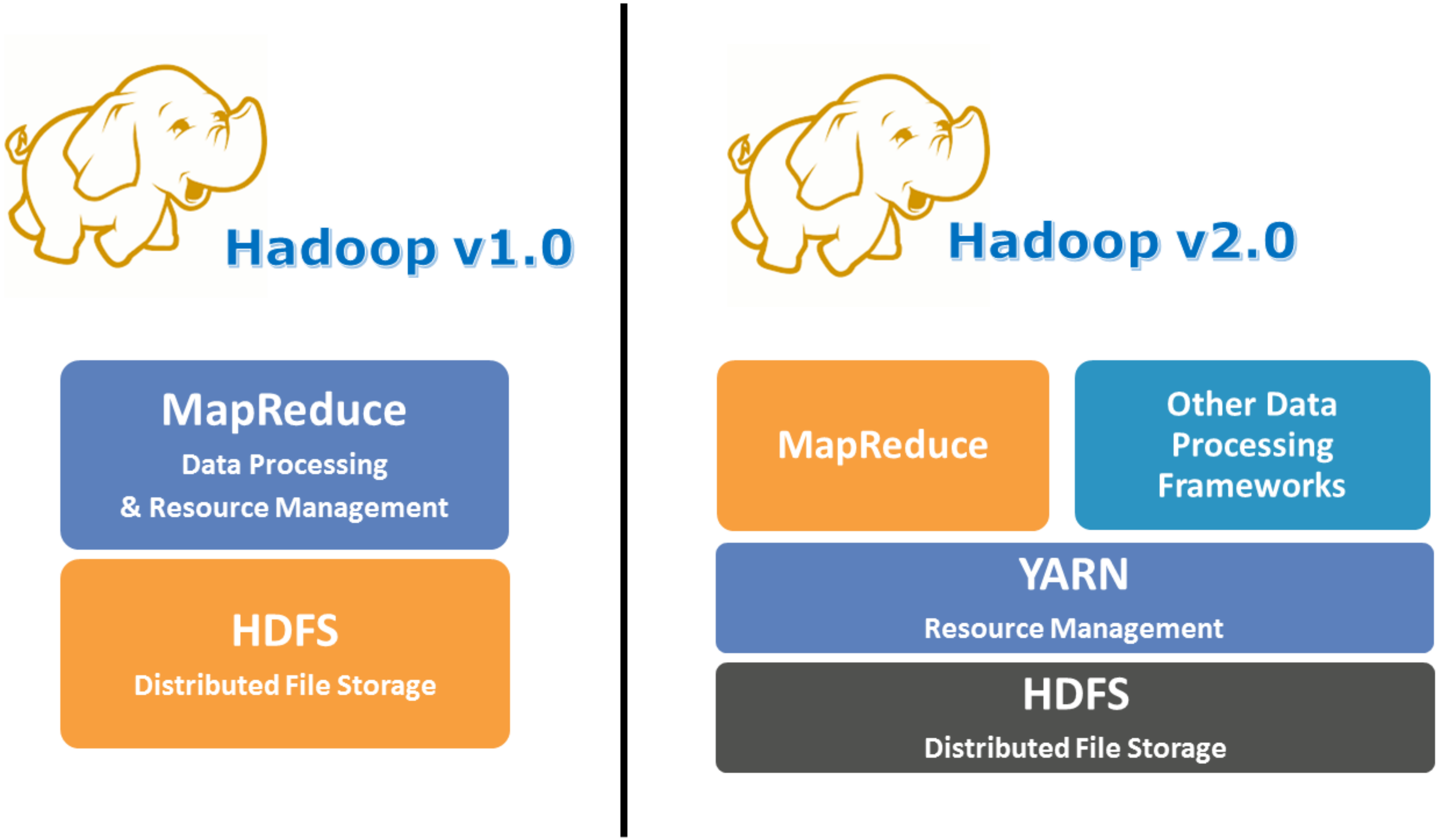
What if we want to run other distributed frameworks?

YARN = Yet-Another-Resource-Negotiator

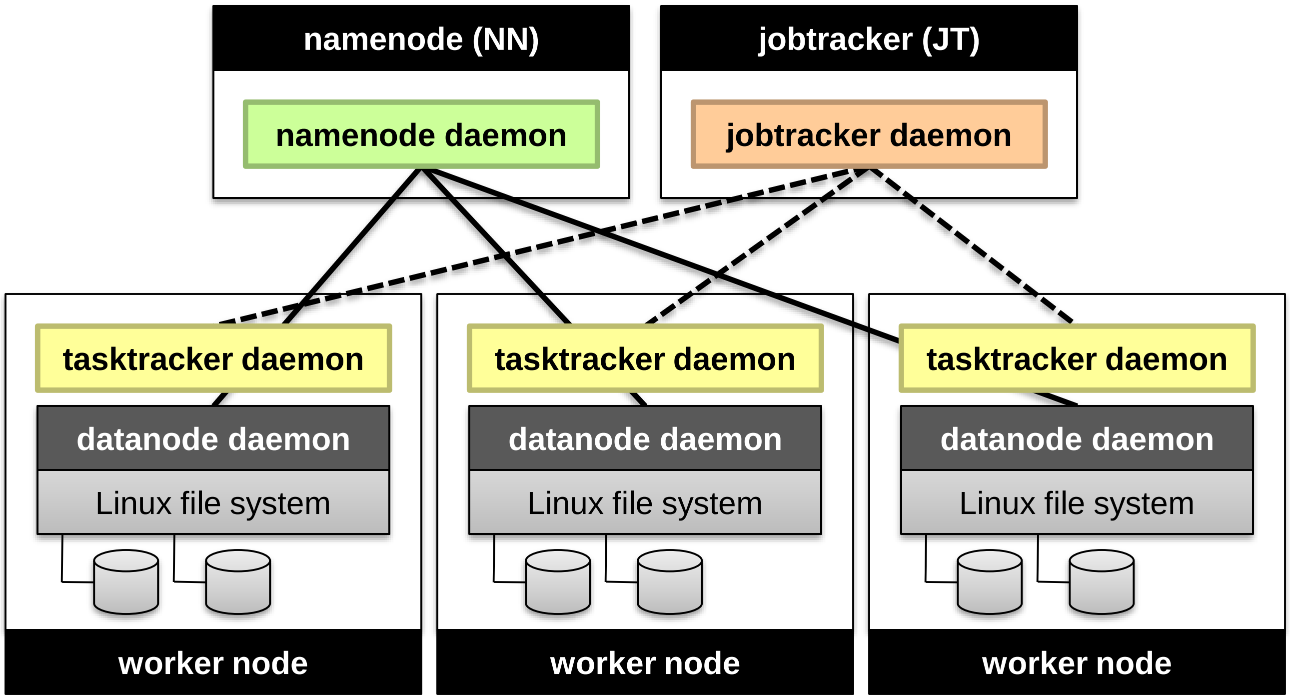
Provides API to develop any generic distributed application

Handles scheduling and resource request

MapReduce (MR2) is one such application in YARN

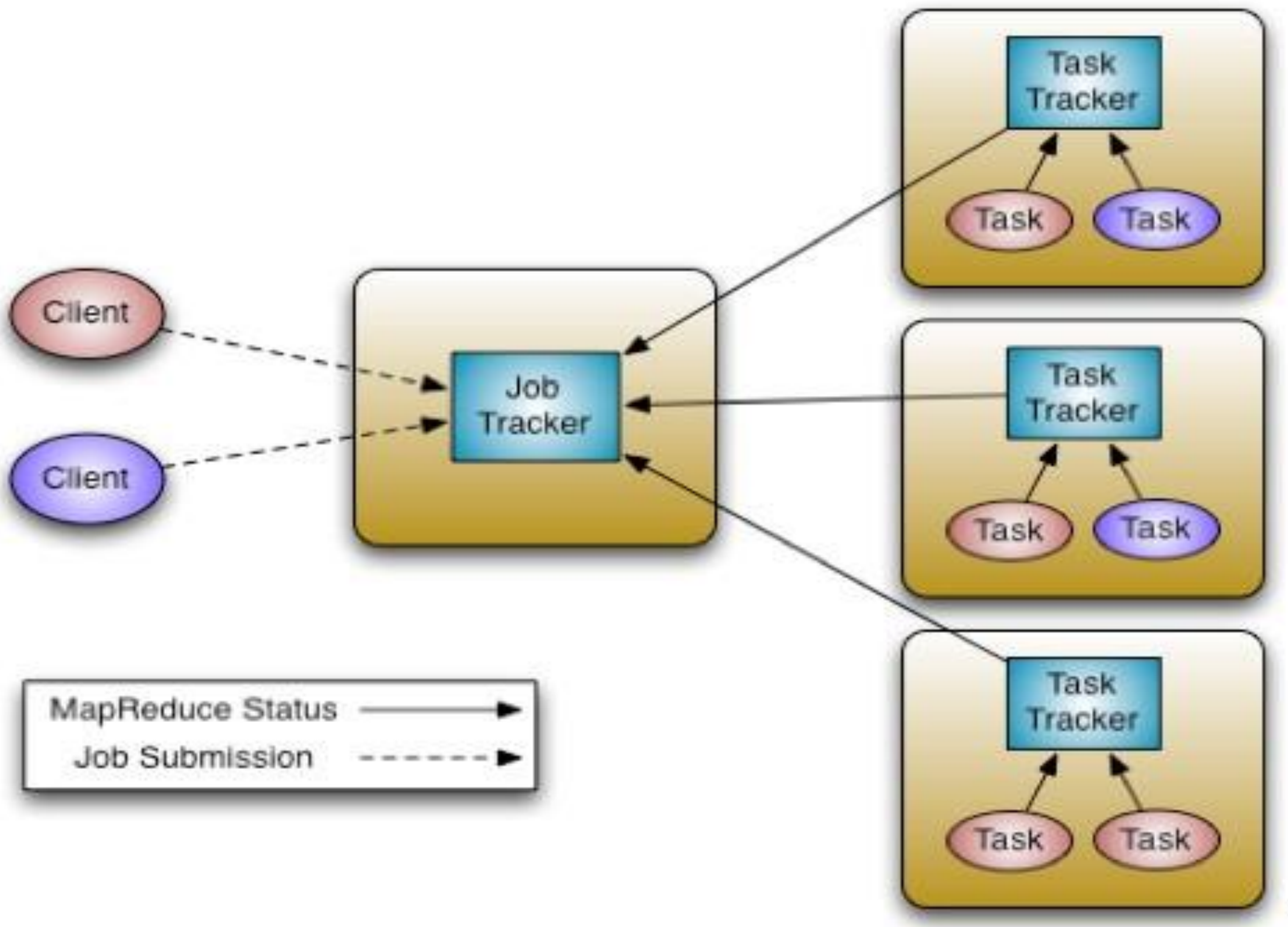

Hadoop v1.0
MapReduce
Data Processing
& Resource Management
HDFS
Distributed File Storage
Hadoop v2.0
MapReduce
Other Data Processing Frameworks
YARN
Resource Management
HDFS
Distributed File Storage

Hadoop MapReduce Architecture

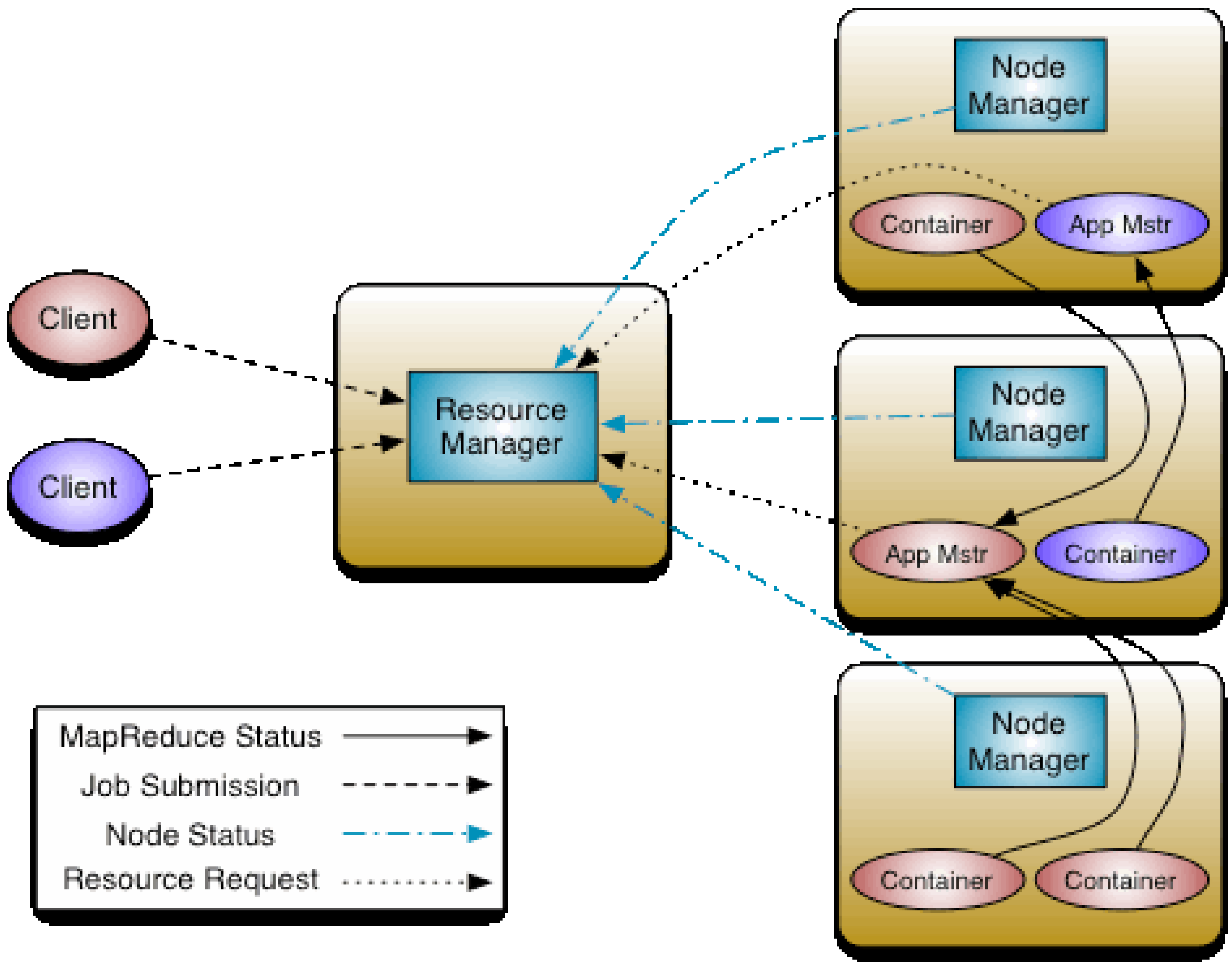


Hadoop v1.0

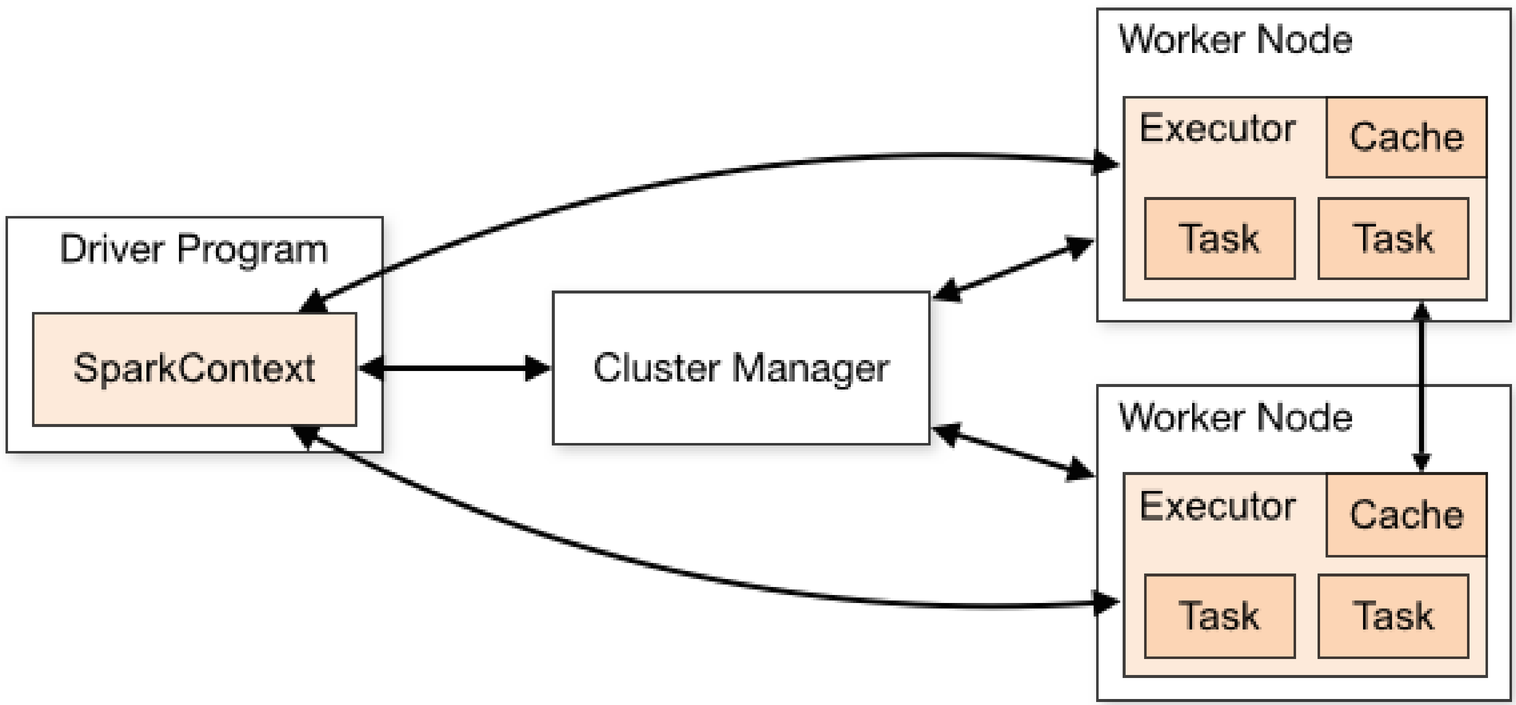
Hadoop v1.0


Task
Tracker
Task
Task
Client
Job
Tracker
Client
Task
Tracker
Task
Task
MapReduce Status
Job Submission
Task
Tracker
Task
Task

Hadoop v2.0


Node Manager
Container
App Mstr
Client
Resource Manager
Node Manager
Client
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

Spark Architecture



Algorithm Design

Closure

Takes type X and returns type X

- 3 + 4 = 7 (int + int = int)
- 5 / 2 = 2.5 (int + int != float)

Identity

“concept of nothing”

- 5 + 0 = 5
- 5 * 1 = 5
- {3, 11, 9} + {} = {3, 11, 9}

- Initializing a counter to zero

Associativity

Add parenthesis anywhere

- 1 + 2 + 3 = (1 + 2) + 3
- 10 / 2 / 5 != 10 / (2 / 5)

- Huge jobs can become many small jobs

Commutativity

Reordering

- $1 + 2 + 3 = 2 + 3 + 1$
- $10 / 2 != 2 / 10$

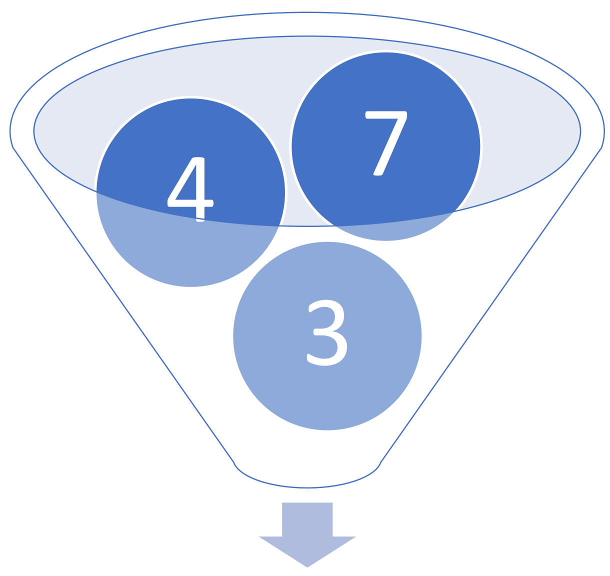
Monoid

- Closure (int + int = int)
- Identity (1 + 0 = 1)
- Associativity (1 + 2 + 3 = (1 + 2) + 3)

- Commutative Monoid

Commutative Monoid and MapReduce

$$\underbrace{(1+1+1)}_{3}+\underbrace{(1+1+1+1+1+1+1)}_{7}+\underbrace{(1+1+1+1)}_{4}$$



14

Two superpowers:

Associativity
Commutativity
(sorting)

Implications for distributed processing?

You don’t know when the tasks begin
You don’t know when the tasks end
You don’t know when the tasks interrupt each other
You don’t know when intermediate data arrive
…

It’s okay!

Word Count: Baseline

```
class Mapper {
  def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
    }
  }
}

class Reducer {
  def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

Computing the Mean: Version 1

```
class Mapper {
  def map(key: String, value: Int) = {
    emit(key, value)
  }
}

class Reducer {
  def reduce(key: String, values: Iterable[Int]) {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    emit(key, sum/cnt)
  }
}
```

Computing the Mean: Version 3

```
class Mapper {
  def map(key: String, value: Int) =
    emit(key, (value, 1))
}
class Combiner {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, sum/cnt)
  }
}
```

Co-occurrence Matrix: Stripes

```
class Mapper {
  def map(key: Long, value: String) = {
    for (u <- tokenize(value)) {
      val map = new Map()
      for (v <- neighbors(u)) {
        map(v) += 1
      }
      emit(u, map)
    }
  }
}

class Reducer {
  def reduce(key: String, values: Iterable[Map]) = {
    val map = new Map()
    for (value <- values) {
      map += value
    }
    emit(key, map)
  }
}
```

Synchronization: Pairs vs. Stripes

Approach 1: turn synchronization into an ordering problem

Sort keys into correct order of computation

Partition key space so each reducer receives appropriate set of partial results

Hold state in reducer across multiple key-value pairs to perform computation

Illustrated by the "pairs" approach

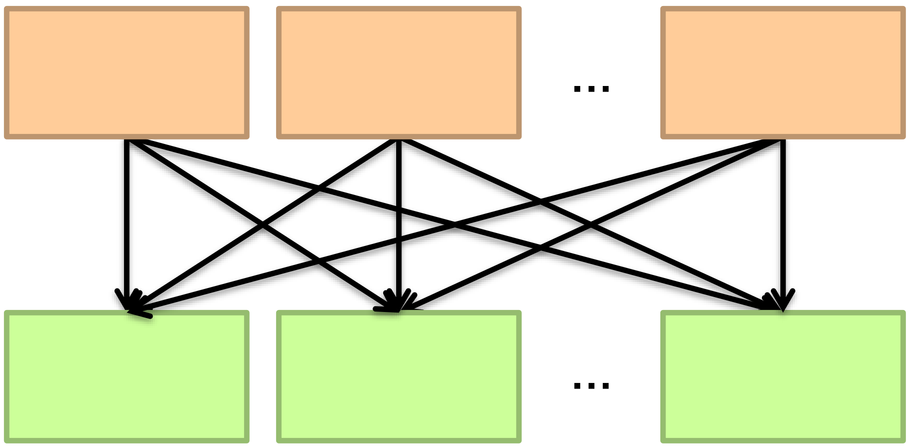
Approach 2: data structures that bring partial results together

Each reducer receives all the data it needs to complete the computation

Illustrated by the "stripes" approach

Commutative monoids!

Because you can't avoid this…



But commutative monoids help

Synchronization: Pairs vs. Stripes

Approach 1: turn synchronization into an ordering problem

Sort keys into correct order of computation

Partition key space so each reducer receives appropriate set of partial results

Hold state in reducer across multiple key-value pairs to perform computation

Illustrated by the "pairs" approach

What about this?

Approach 2: data structures that bring partial results together

Each reducer receives all the data it needs to complete the computation

Illustrated by the "stripes" approach

Commutative monoids!

$f(B|A)$: "Pairs"

$(a, *) \rightarrow 32$ Reducer holds this value in memory

$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$	$(a, b_4) \rightarrow 1 / 32$
…	…

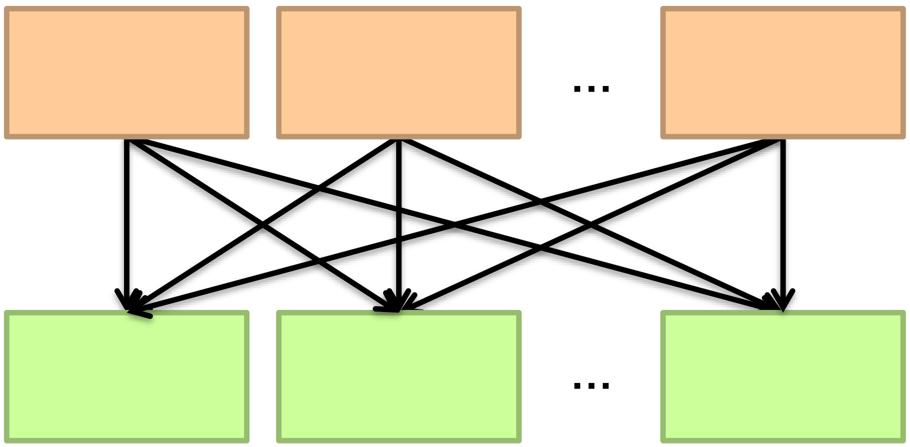
For this to work:

Emit extra $(a, *)$ for every b_n in mapper
Make sure all a's get sent to same reducer (use partitioner)
Make sure $(a, *)$ comes first (define sort order)
Hold state in reducer across different key-value pairs

Two superpowers:

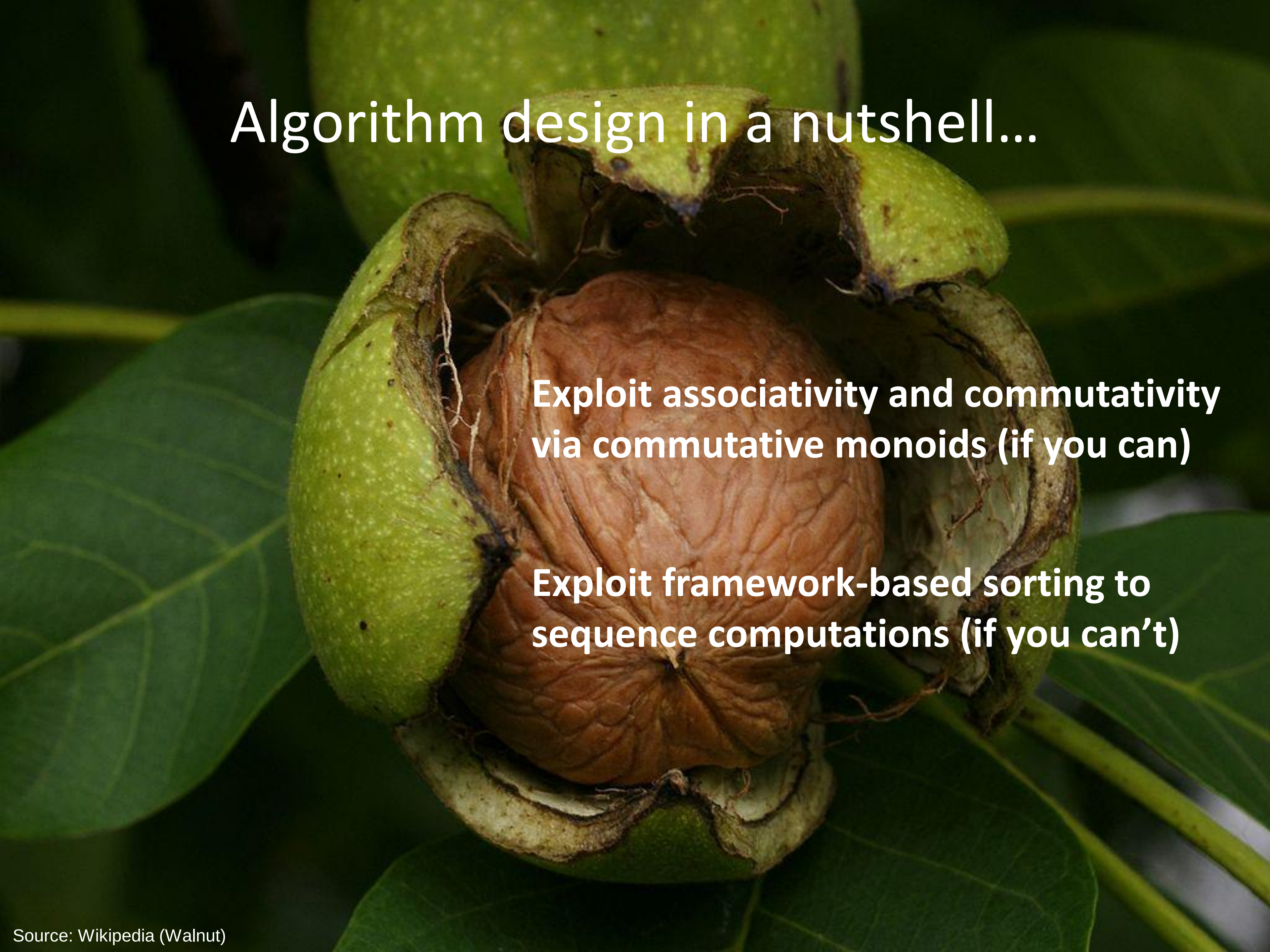
Associativity
Commutativity
(sorting)

When you can't "monoidify"



Sequence your computations by sorting

Algorithm design in a nutshell…



Exploit associativity and commutativity via commutative monoids (if you can)

Exploit framework-based sorting to sequence computations (if you can't)

Source: Wikipedia (Walnut)