ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Reduction Operations

Big Data Analysis with Scala and Spark

Heather Miller

What we've seen so far

- we defined *Distributed Data Parallelism*
- we saw that Apache Spark implements this model
- we got a feel for what latency means to distributed systems

What we've seen so far

- we defined *Distributed Data Parallelism*
- we saw that Apache Spark implements this model
- we got a feel for what latency means to distributed systems

Spark's Programming Model

- We saw that, at a glance, Spark looks like Scala collections
- However, internally, Spark behaves differently than Scala collections
  - Spark uses ***laziness*** to save time and memory
- We saw *transformations* and *actions*
- We saw caching and persistence (*i.e.*, cache in memory, save time!)
- We saw how the cluster topology comes into the programming model

Transformations to Actions

Most of our intuitions have focused on distributing **transformations** such as `map`, `flatMap`, `filter`, etc.

We've visualized how transformations like these are distributed and parallelized.

Transformations to Actions

Most of our intuitions have focused on distributing **transformations** such as `map`, `flatMap`, `filter`, etc.

We've visualized how transformations like these are distributed and parallelized.

But what about actions? In particular, how are common `reduce`-like actions distributed in Spark?

Reduction Operations, Generally

First, what do we mean by "reduction operations"?

Recall operations such as `fold`, `reduce`, and `aggregate` from Scala sequential collections. All of these operations and their variants (such as `foldLeft`, `reduceRight`, etc) have something in common.

Reduction Operations, Generally

First, what do we mean by “reduction operations”?

Recall operations such as `fold`, `reduce`, and `aggregate` from Scala sequential collections. All of these operations and their variants (such as `foldLeft`, `reduceRight`, etc) have something in common.

Reduction Operations:
walk though a collection and combine neigboring elements of the collection together to produce a single combined result.
(rather than another collection)

Reduction Operations, Generally

Reduction Operations:

walk though a collection and combine neighboring elements of the collection together to produce a single combined result.

(rather than another collection)

Example:

```
case class Taco(kind: String, price: Double)

val tacoOrder =
  List(
    Taco("Carnitas", 2.25),
    Taco("Corn", 1.75),
    Taco("Barbacoa", 2.50),
    Taco("Chicken", 2.00))

val cost = tacoOrder.foldLeft(0.0)((sum, taco) => sum + taco.price)
```

Parallel Reduction Operations

Recall what we learned in the course Parallel Programming course about `foldLeft` vs `fold`.

Which of these two were parallelizable?

Parallel Reduction Operations

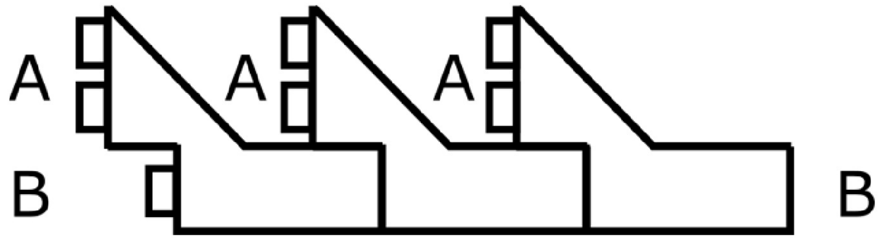
Recall what we learned in the course Parallel Programming course about `foldLeft` vs `fold`.

Which of these two were parallelizable?

`foldLeft` is not parallelizable.

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```

> *Applies a binary operator to a start value and all elements of this collection or iterator, going left to right.*
>
> — Scala API documentation



Parallel Reduction Operations: FoldLeft

`foldLeft` is not parallelizable.

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

Being able to change the result type from `A` to `B` forces us to have to execute `foldLeft` sequentially from left to right.

Concretely, given:

"1234"

```
val xs = List(1, 2, 3, 4)
val res = xs.foldLeft(””)((str: String, i: Int) => str + i)
```

What happens if we try to break this collection in two and parallelize?

Parallel Reduction Operations: FoldLeft

foldLeft is not parallelizable.

```
def foldLeft[B](z: B)(f: (B, A) => B): B

val xs = List(1, 2, 3, 4)
val res = xs.foldLeft("")((str: String, i: Int) => str + i)
```

String

List(1, 2)

"" + 1 → "1"
"1" + 2 → "12"
string

List(3, 4)

"" + 3 → "3"
"3" + 4 → "34"
String

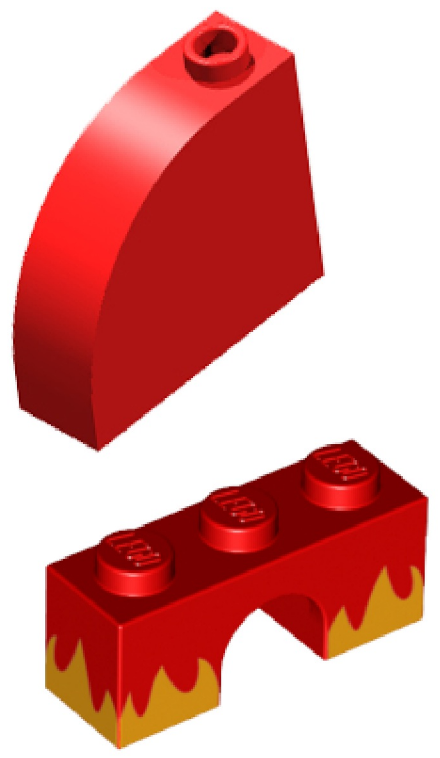
!! type error !!

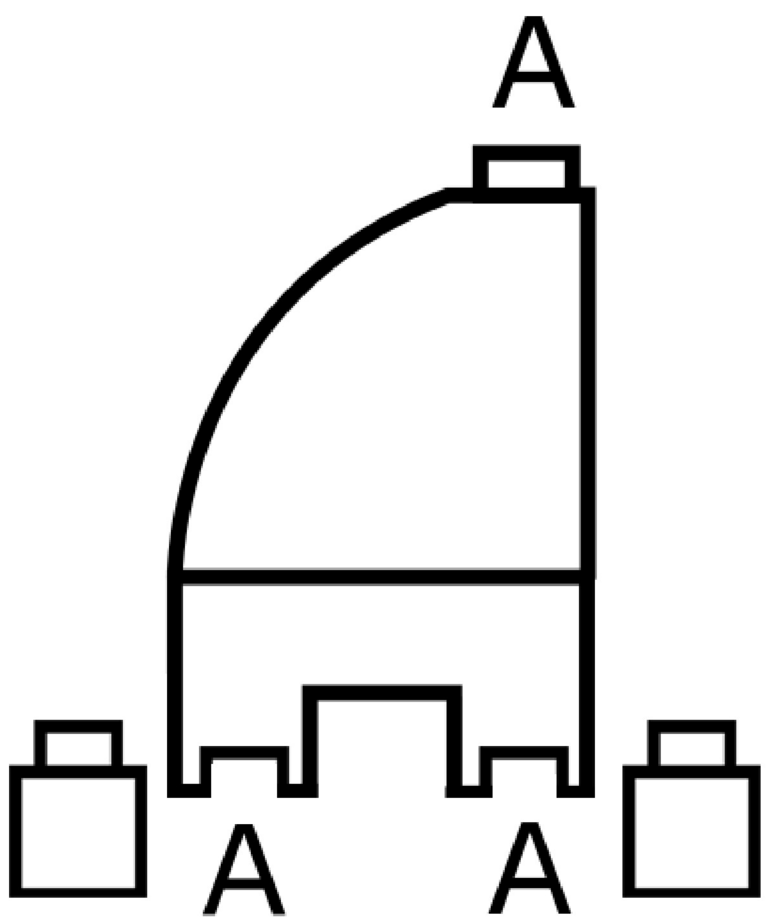
can't apply (str: String, i: Int) => str + i !!

Parallel Reduction Operations: Fold

`fold` enables us to parallelize things, but it restricts us to always returning the same type.

```scala
def fold(z: A)(f: (A, A) => A): A
```



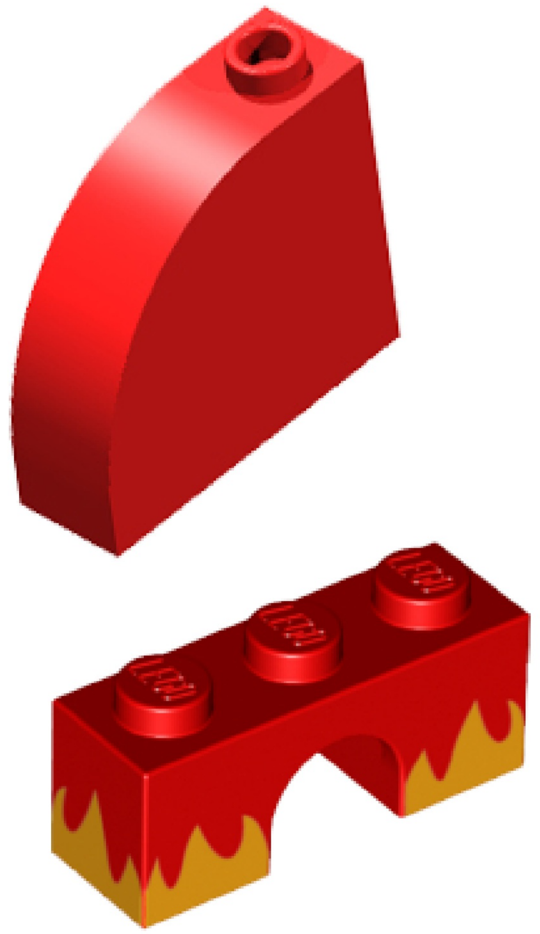


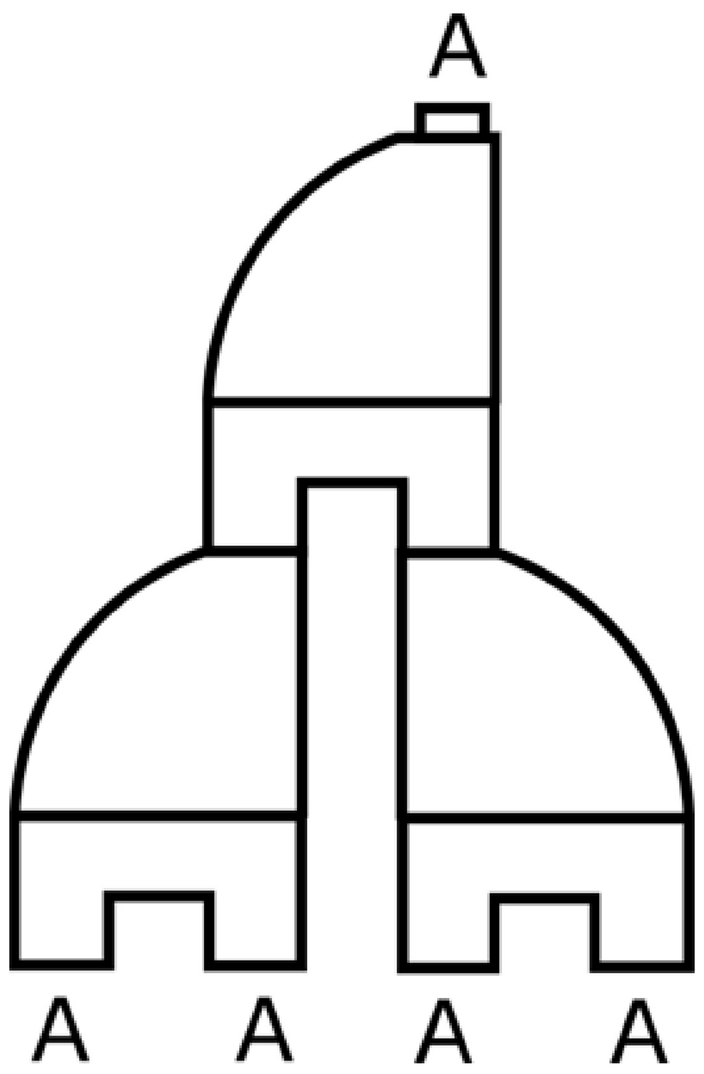
It enables us to parallelize using a single function `f` by enabling us to build parallelizable reduce trees.

Parallel Reduction Operations: Fold

It enables us to parallelize using a single function `f` by enabling us to build parallelizable reduce trees.

```
def fold(z: A)(f: (A, A) => A): A
```





Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

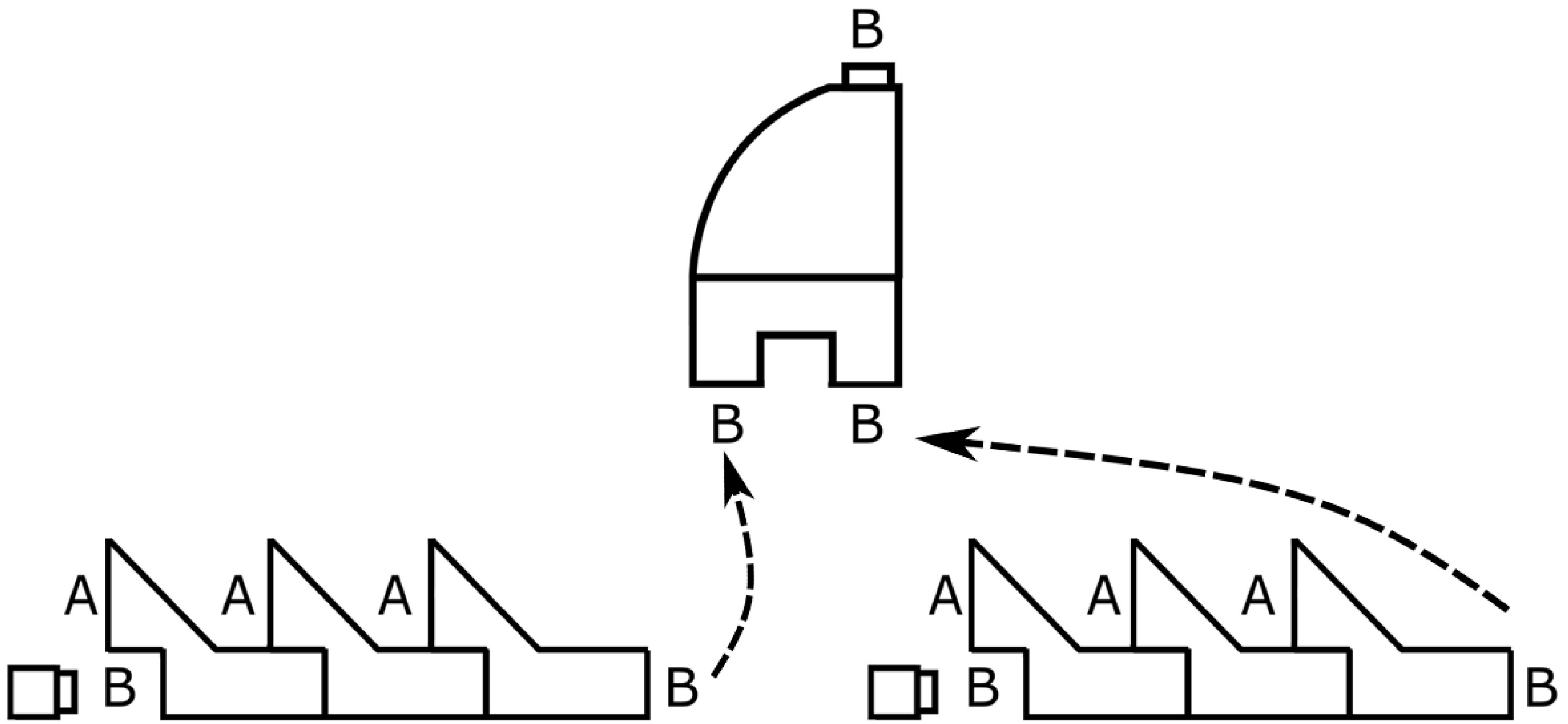
`aggregate` is said to be general because it gets you the best of both worlds.

Properties of `aggregate`

1. Parallelizable.
2. Possible to change the return type.

Parallel Reduction Operations: Aggregate

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```



Aggregate lets you still do sequential-style folds *in chunks* which change the result type. Additionally requiring the `combop` function enables building one of these nice reduce trees that we saw is possible with `fold` to *combine these chunks* in parallel.

Reduction Operations on RDDs

Scala collections:

fold

foldLeft/foldRight

reduce

aggregate

Spark:

fold

~~foldLeft/foldRight~~

reduce

aggregate

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

Spark:
fold
~~foldLeft/foldRight~~
reduce
aggregate

Spark doesn't even give you the option to use `foldLeft`/`foldRight`. Which means that if you have to change the return type of your reduction operation, your only choice is to use `aggregate`.

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

Spark:
fold
~~foldLeft/foldRight~~
reduce
aggregate

Spark doesn't even give you the option to use `foldLeft/foldRight`. Which means that if you have to change the return type of your reduction operation, your only choice is to use `aggregate`.

Question: *Why not still have a serial foldLeft/foldRight on Spark?*

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

Spark:
fold
~~foldLeft/foldRight~~
reduce
aggregate

Spark doesn't even give you the option to use `foldLeft/foldRight`. Which means that if you have to change the return type of your reduction operation, your only choice is to use `aggregate`.

Question: *Why not still have a serial foldLeft/foldRight on Spark?*

Doing things serially across a cluster is actually difficult. Lots of synchronization. Doesn't make a lot of sense.

RDD Reduction Operations: Aggregate

In Spark, `aggregate` is a more desirable reduction operator a majority of the time. Why do you think that's the case?

RDD Reduction Operations: Aggregate

In Spark, `aggregate` is a more desirable reduction operator a majority of the time. Why do you think that's the case?

As you will realize from experimenting with our Spark ~~cluster~~ assignments, much of the time when working with large-scale data, our goal is to ***project down from larger/more complex data types***.

RDD Reduction Operations: Aggregate

In Spark, `aggregate` is a more desirable reduction operator a majority of the time. Why do you think that's the case?

As you will realize from experimenting with our Spark cluster, much of the time when working with large-scale data, our goal is to ***project down from larger/more complex data types***.

Example:

```
case class WikipediaPage(
  title: String,
  redirectTitle: String,
  timestamp: String,
  lastContributorUsername: String,
  text: String)
```

RDD Reduction Operations: Aggregate

As you will realize after experimenting with Spark a bit, much of the time when working with large-scale data, your goal is to ***project down from larger/more complex data types***.

Example:

```scala
case class WikipediaPage(
  title: String,
  redirectTitle: String,
  timestamp: String,
  lastContributorUsername: String,
  text: String)
```

I might only care about `title` and `timestamp`, for example. In this case, it'd save a lot of time/memory to not have to carry around the full-text of each article (`text`) in our accumulator!

Hence, why `accumulate` is often more desirable in Spark than in Scala collections!