CSEP 524 – Parallel Computation University of Washington

Lecture 8: Data Analytics II/Spark

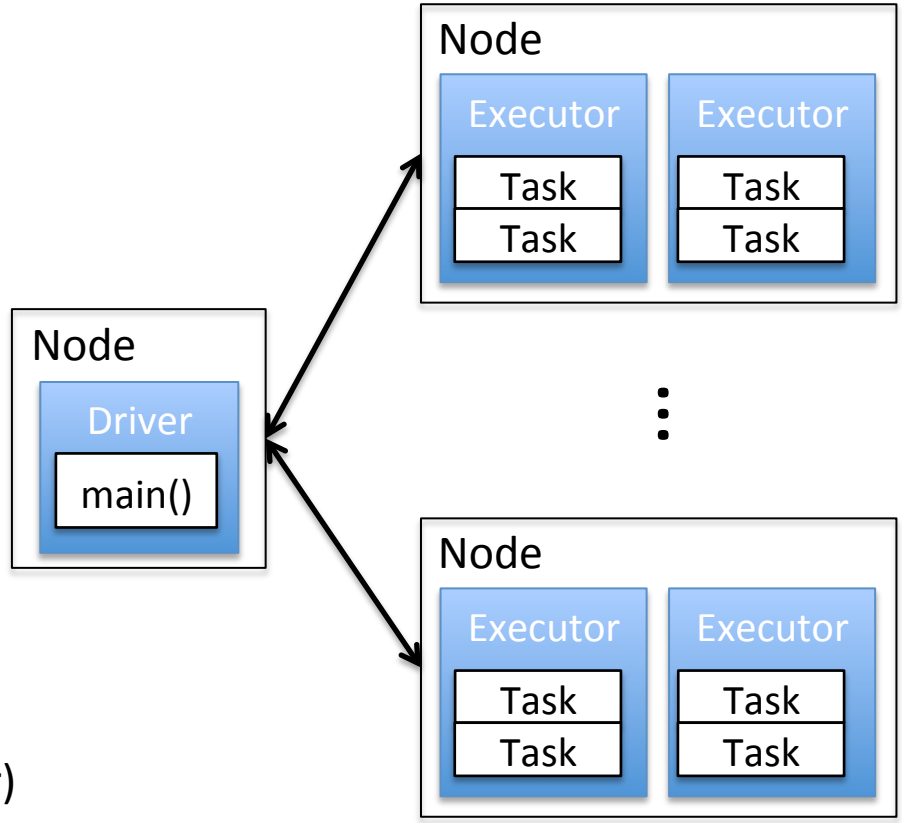
Michael Ringenburg
Spring 2015

Logistics

- First few presentations at end of class today
- Rest of you will be during the next two classes
  - Please send me your slides the night before
- Today is the last lecture … Continueing our data analytics exploration with a deep dive on Spark

Spark Execution Model

- "Master-slave" parallelism model
- Driver (master)
  - Executes main
  - Distributes data and work to executors
- Resilient Distributed Dataset (RDD)
  - Spark's primary data abstraction
  - Partitioned amongst executors
  - Fault-tolerant via lineage
- Executors (slaves)
  - Lazily execute tasks (operations on partitions of the RDD)
  - Global all-to-all shuffle (with barrier) for data exchange



RDD In Depth

- Primary data abstraction of Spark
  - (Prior to new DataFrames in 1.3)
- Five parts (two of which are optional)
  - Set of partitions
  - List of dependencies ("parent RDDs")
  - Function to compute my partition from my parents
  - Method of compute partitioning of data (optional)
  - Preferred location for each partition (e.g., HDFS block location)
- Notice that the RDDs contain a description of the data and computation, but not the actual data … We will see why soon …

Spark Programming: Simple Example


```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Spark Programming: Simple Example

Create array of {1, 2, …, 1,000,000}

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Spark Programming: Simple Example

Create array of {1, 2, …, 1,000,000}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Spark Programming: Simple Example

Create array of {1, 2, …, 1,000,000}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

Filter: example of a Spark *transformation* (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Spark Programming: Simple Example

Create array of {1, 2, …, 1,000,000}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

Filter: example of a Spark *transformation* (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

Spark *action* (return result to driver)

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

Spark Programming: Simple Example

Create array of {1, 2, …, 1,000,000}

Partition array into a 8-partition RDD distributed across executor nodes. (Can also create from file.)

Filter: example of a Spark *transformation* (create new RDD from old RDD). Filter keeps data for which the argument evaluates to true.

Spark *action* (return result to driver)

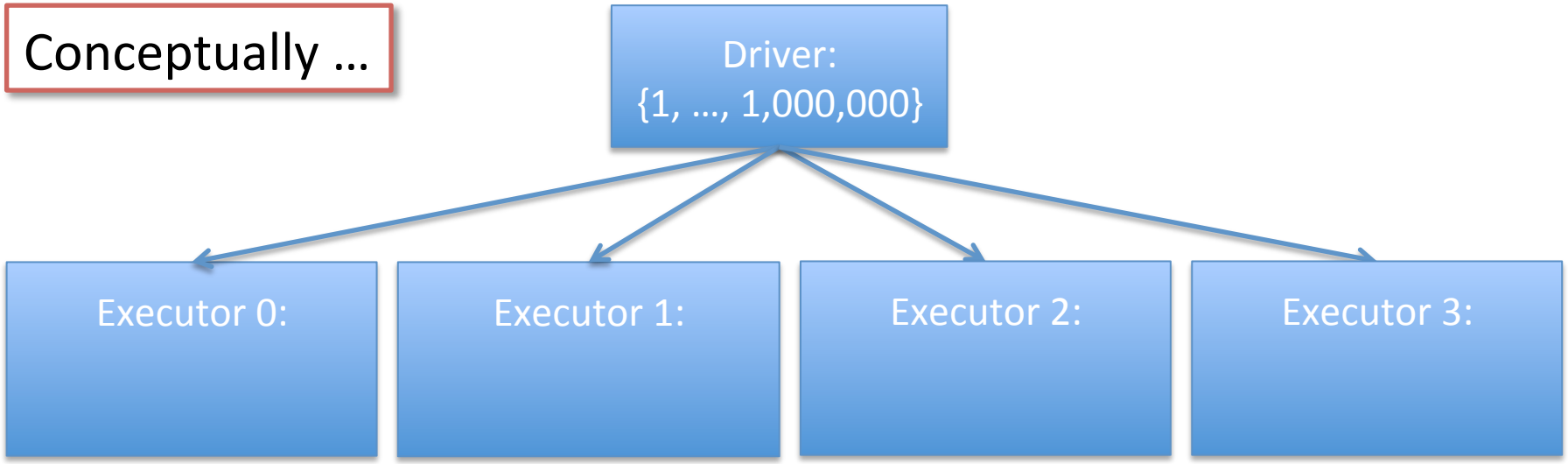
```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(
              a => (a%2) == 0
            )
evens.take(5)

>>> Array[Int] = Array(2, 4, 6, 8, 10)
```

compute

Lazy Evaluation: No computation until result requested

Example: Line-by-line

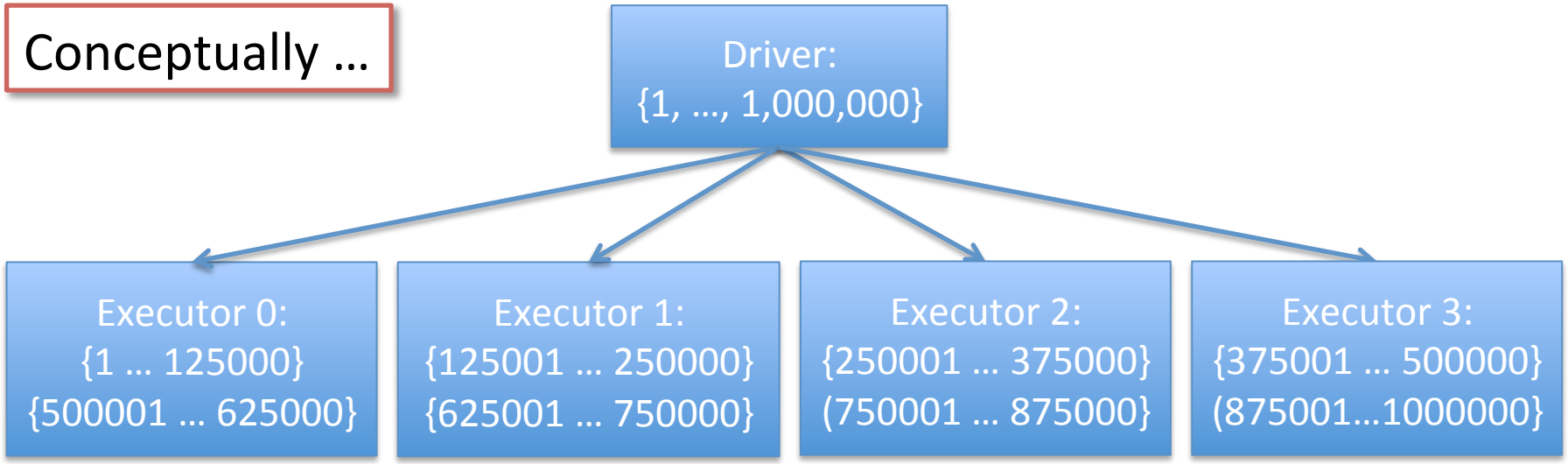
Conceptually …




```
val arr1M = Array.range(1,1000001)
```

Example: Line-by-line

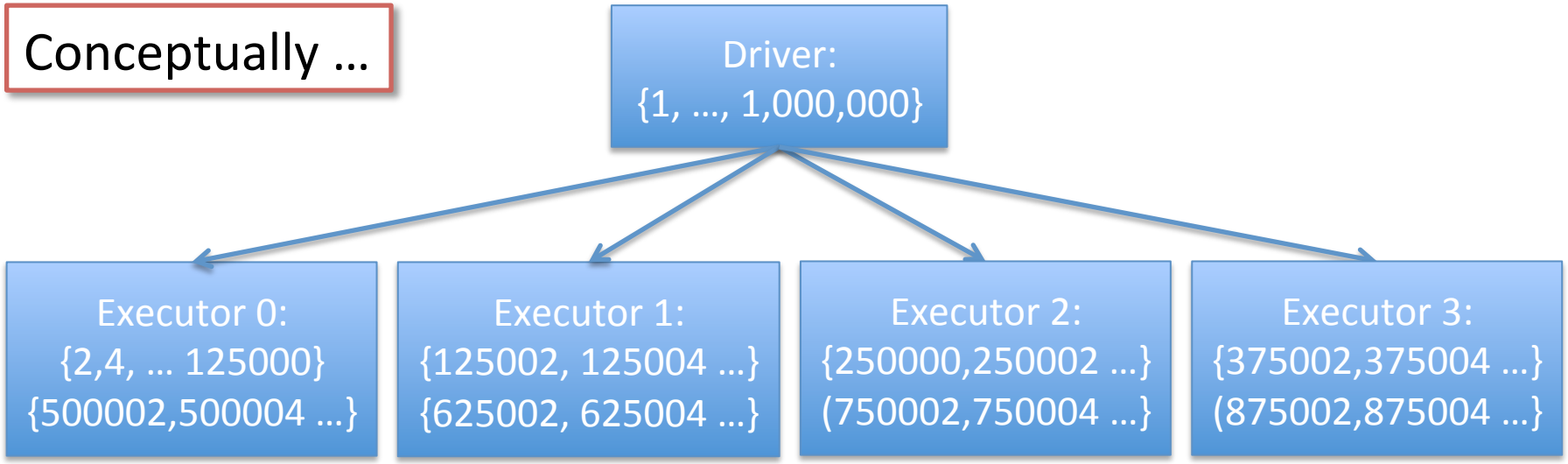
Conceptually …




```
val rdd1M = sc.parallelize(arr1M, 8)
```

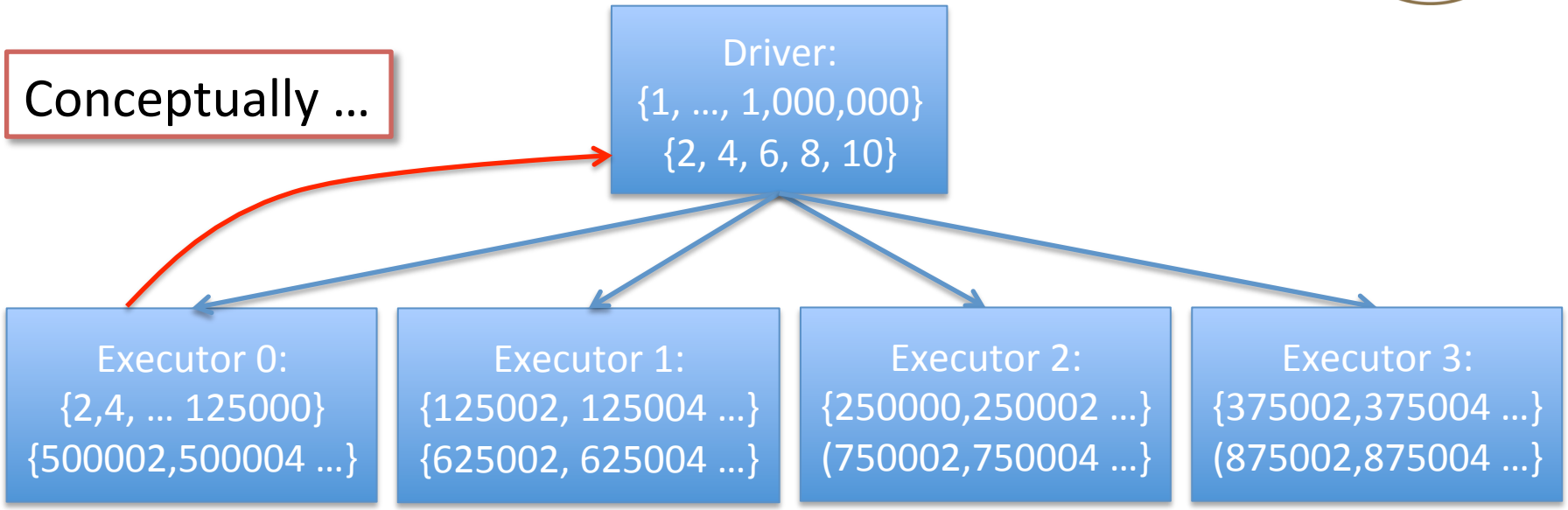
Example: Line-by-line

Conceptually …




```
val evens = rdd1M.filter(a => a%2==0)
```

Example: Line-by-line



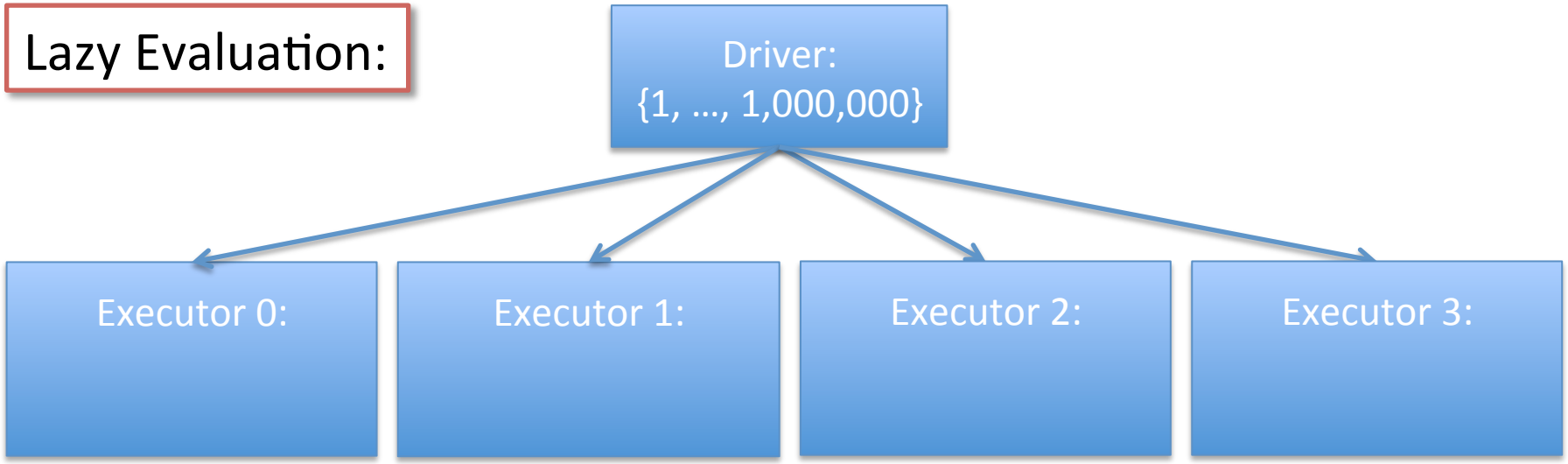

```
evens.take(5)
```

Example: Line-by-line

Now let's try it out …

What's going on here?

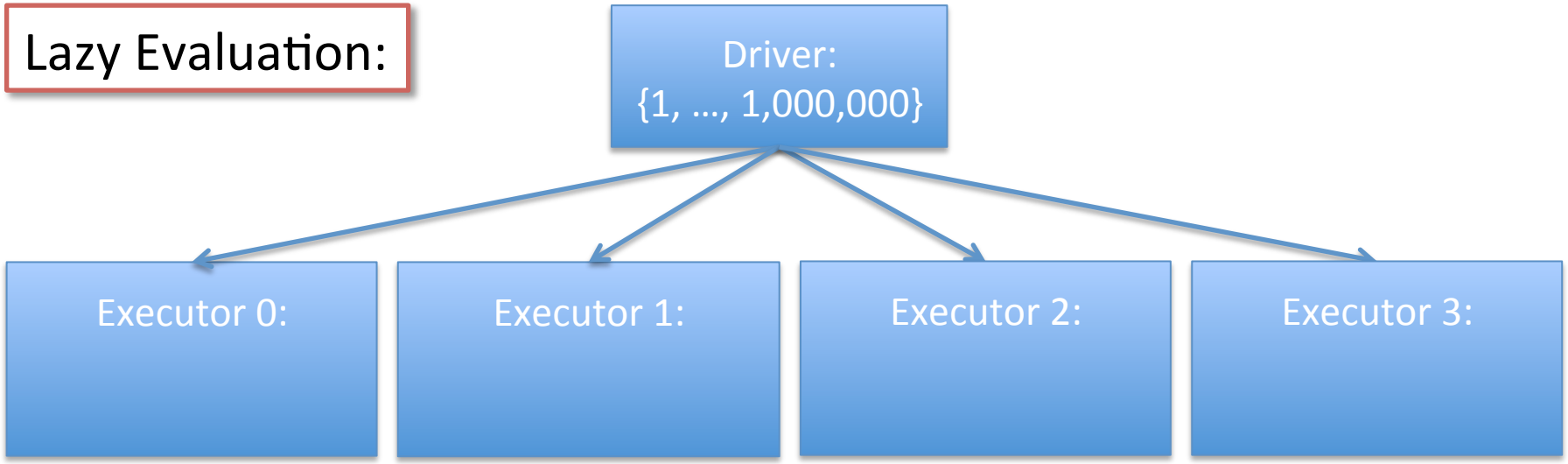
Lazy Evaluation:

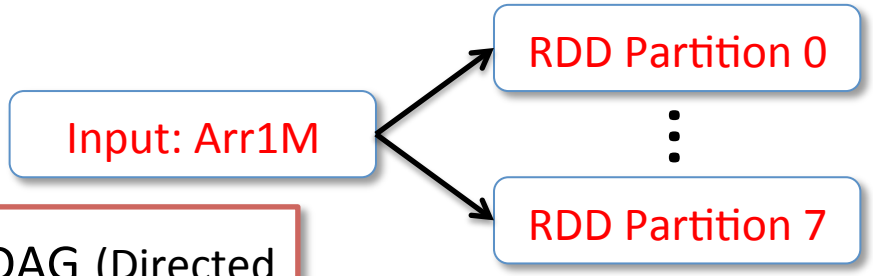



```
val arr1M = Array.range(1,1000001)
```

What's going on here?

Lazy Evaluation:



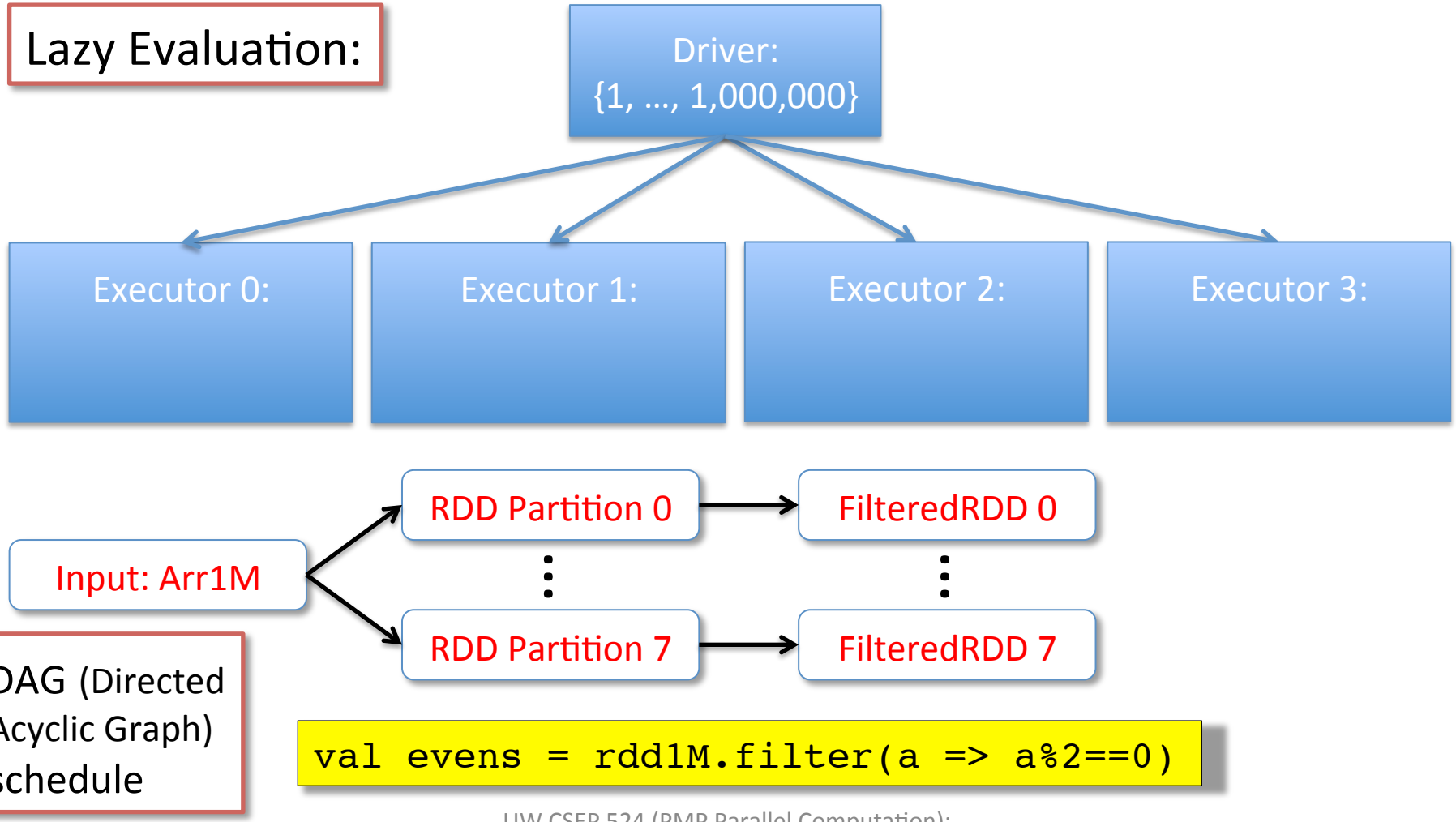


DAG (Directed Acyclic Graph) schedule

```
val rdd1M = sc.parallelize(arr1M, 8)
```

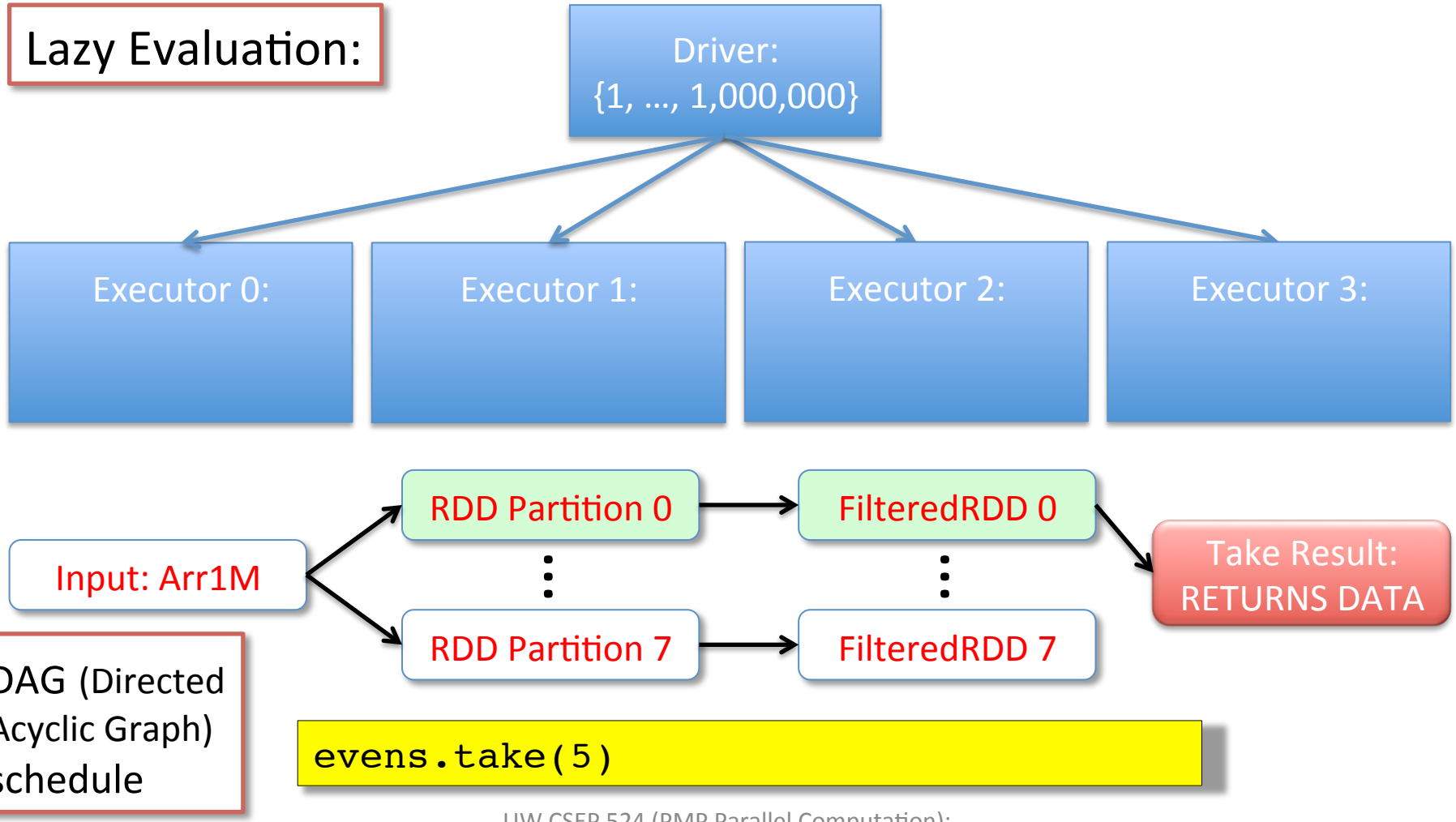
What's going on here?

Lazy Evaluation:



What's going on here?

Lazy Evaluation:



What's going on here?

Lazy Evaluation:

Start computing!

Driver:
{1, …, 1,000,000}

Executor 0:
{1 … 125000}

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

RDD Partition 7

FilteredRDD 0

FilteredRDD 7

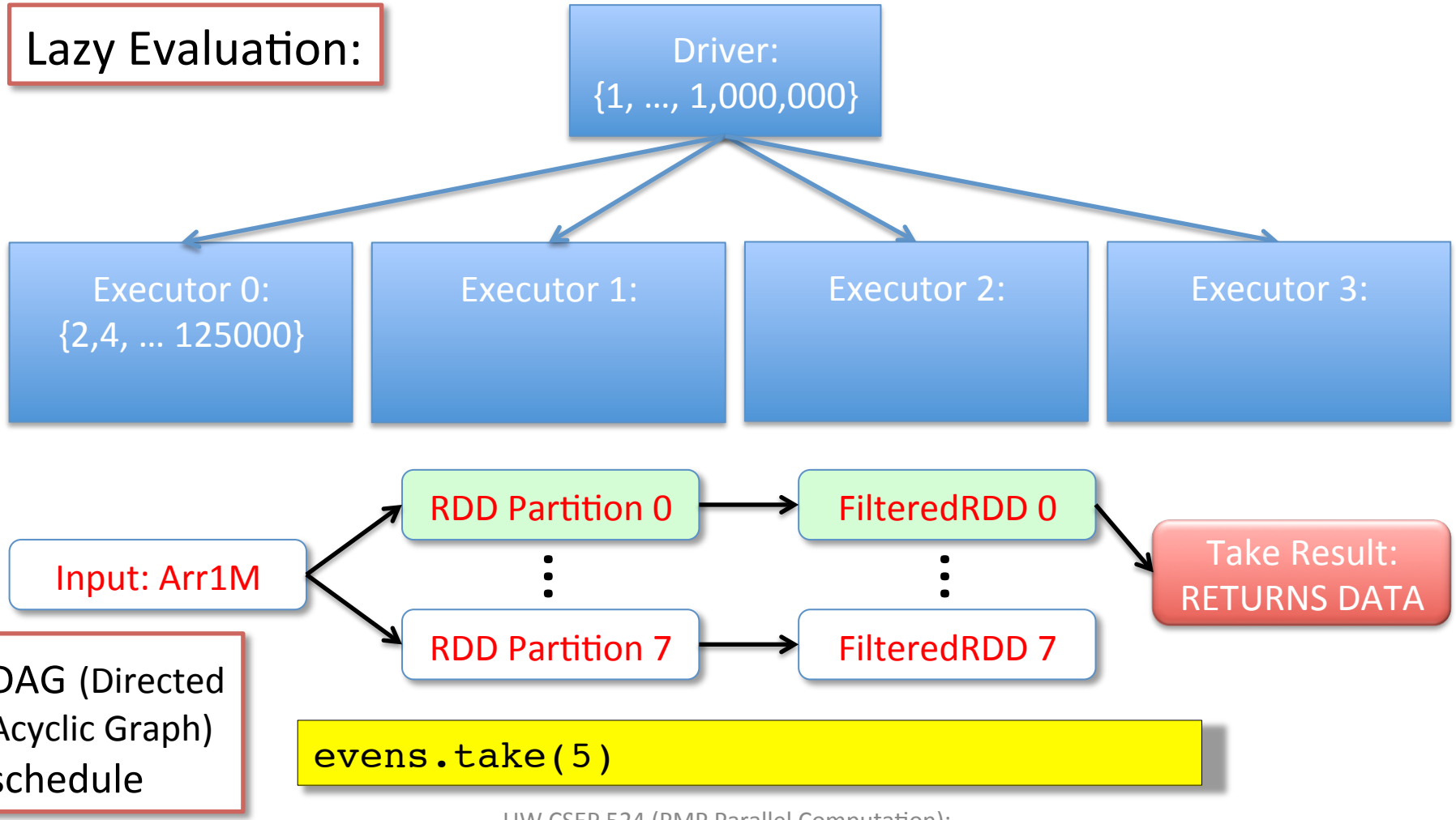
Take Result:
RETURNS DATA

DAG (Directed Acyclic Graph) schedule

```
evens.take(5)
```

What's going on here?

Lazy Evaluation:



What's going on here?

Lazy Evaluation:

Driver:
{1, …, 1,000,000}
{2, 4, 6, 8, 10}

Executor 0:
{2,4, … 125000}

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

RDD Partition 0

RDD Partition 7

FilteredRDD 0

FilteredRDD 7

Take Result:
RETURNS DATA

DAG (Directed Acyclic Graph) schedule

```
evens.take(5)
```

Modified example


```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Imagine we want to perform a number of operations on our filtered RDD of even integers.
- For each action, Spark will compute the DAG steps…

Modified example

Count returns the total size (# elems) of an RDD.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Imagine we want to perform a number of operations on our filtered RDD of even integers.
- For each action, Spark will compute the DAG steps…

Modified example

Count returns the total size (# elems) of an RDD.

Reduce performs a reduction over the dataset, combining elements with the argument function.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Imagine we want to perform a number of operations on our filtered RDD of even integers.
- For each action, Spark will compute the DAG steps…

Modified example

Count returns the total size (# elems) of an RDD.

Reduce performs a reduction over the dataset, combining elements with the argument function.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Problem: This means recomputing the filtered "evens" RDD three times – inefficient.

Modified example

Persist tells Spark to keep the data in memory even after it is done with the action. Allows future actions to reuse without recomputing. Cache is synonym for default storage level (memory). Can also persist on disk, etc.

```
val arr1M = Array.range(1,1000001)
val rdd1M = sc.parallelize(arr1M, 8)
val evens = rdd1M.filter(a => (a%2) == 0)
evens.persist()  // or cache()
val firstFiveEvens = evens.take(5)
// How many evens?
val totalEvens = evens.count()
// Sum of evens
val evenSum = evens.reduce((a,b) => a+b)
```

- Problem: This means recomputing the filtered "evens" RDD three times – inefficient.
- Solution: Persist the RDD!

Modified example

Demo…

Communication Example


```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

Communication Example

Load file, default number of paritions is # of HDFS blocks

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

Communication Example

Load file, default number of paritions is # of HDFS blocks

flatMap maps one value to (possibly) many, instead of one-to-one like map

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

Communication Example

Load file, default number of paritions is # of HDFS blocks

flatMap maps one value to (possibly) many, instead of one-to-one like map

groupByKey combines all key-value pairs with the same key (k, v1), …, (k,vn) into a single key-value pair (k, (v1, …, vn)).

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

Communication Example

Load file, default number of paritions is # of HDFS blocks

flatMap maps one value to (possibly) many, instead of one-to-one like map

groupByKey combines all key-value pairs with the same key (k, v1), …, (k,vn) into a single key-value pair (k, (v1, …, vn)).

Collect returns all elements to the driver

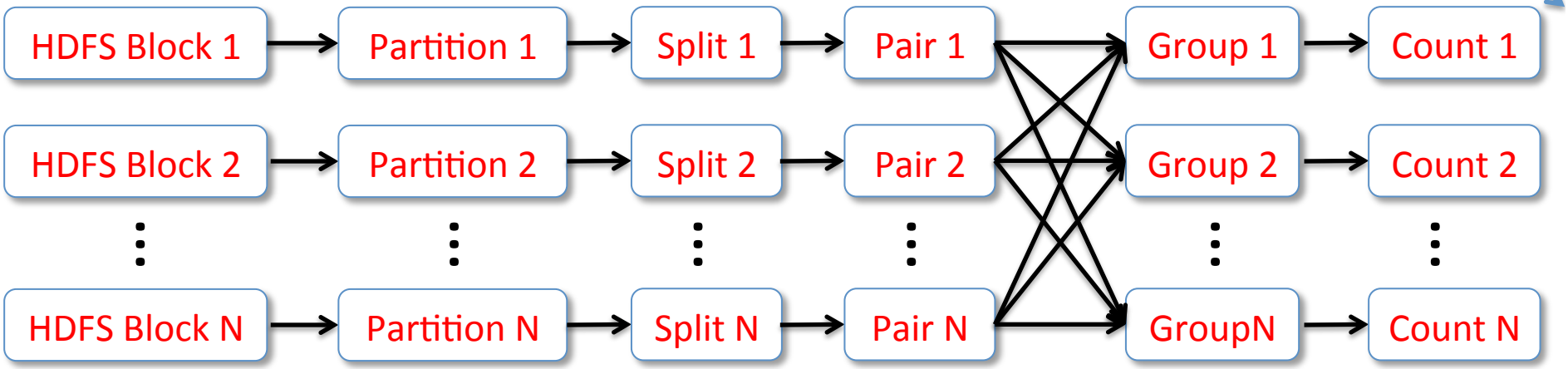
```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

- Let's like at a global communication example: computing the number of times each word occurs
  - Load a text file
  - Split it into words
  - Group same words together (all-to-all communication)
  - Count each word

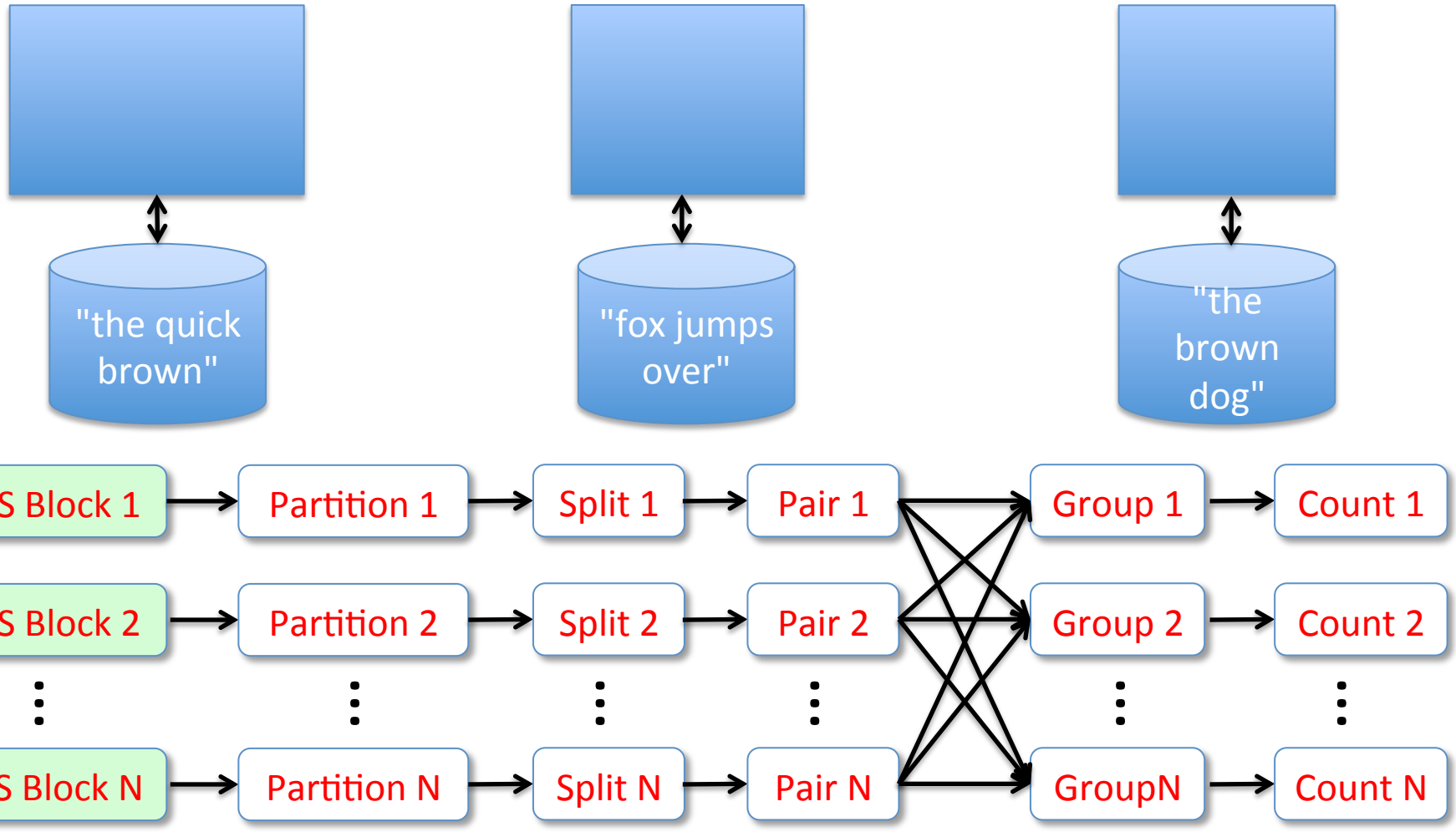
The DAG

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts = wordCounts.collect()
```

Omitting collect due to space constraints

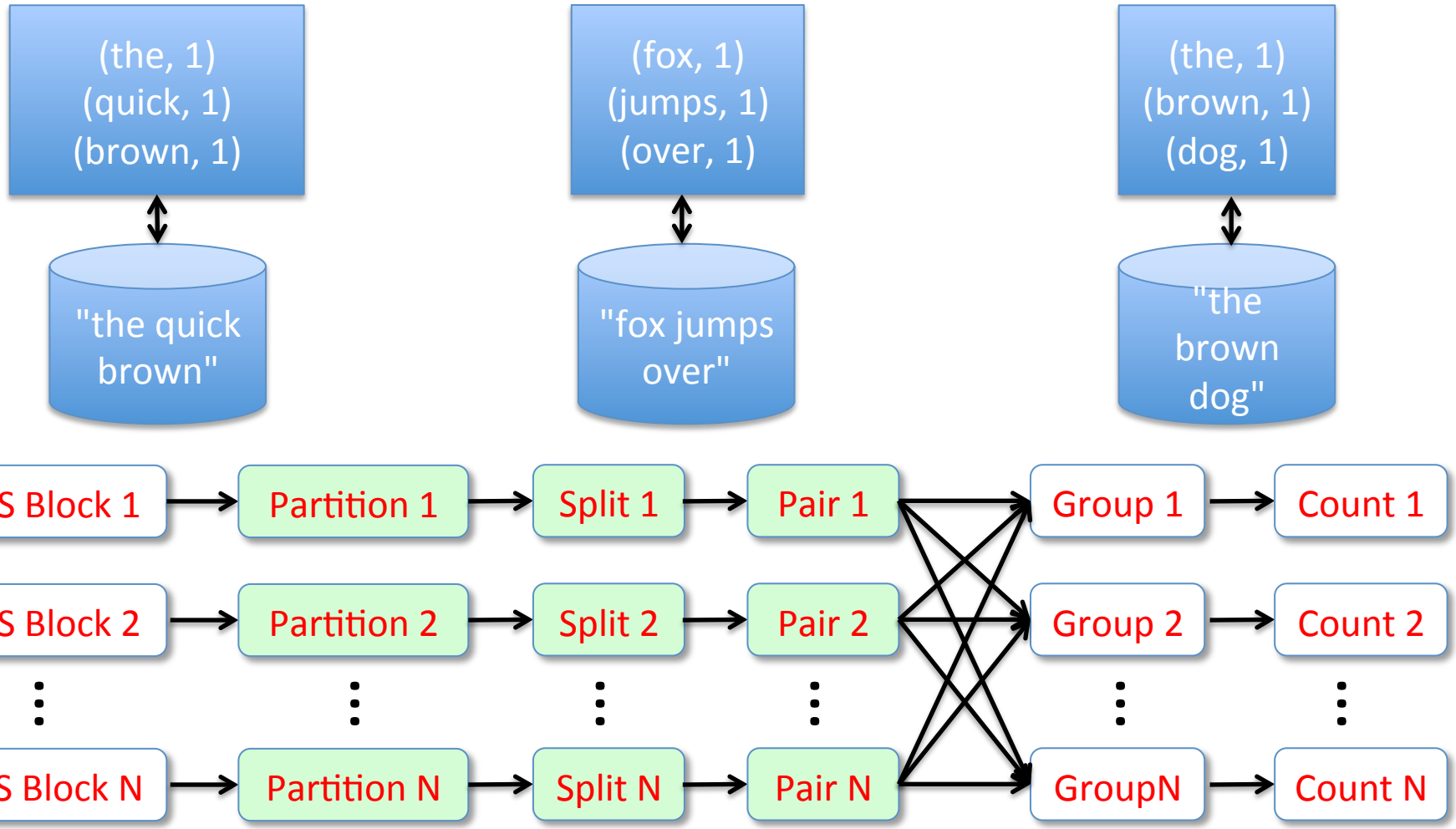


Execution

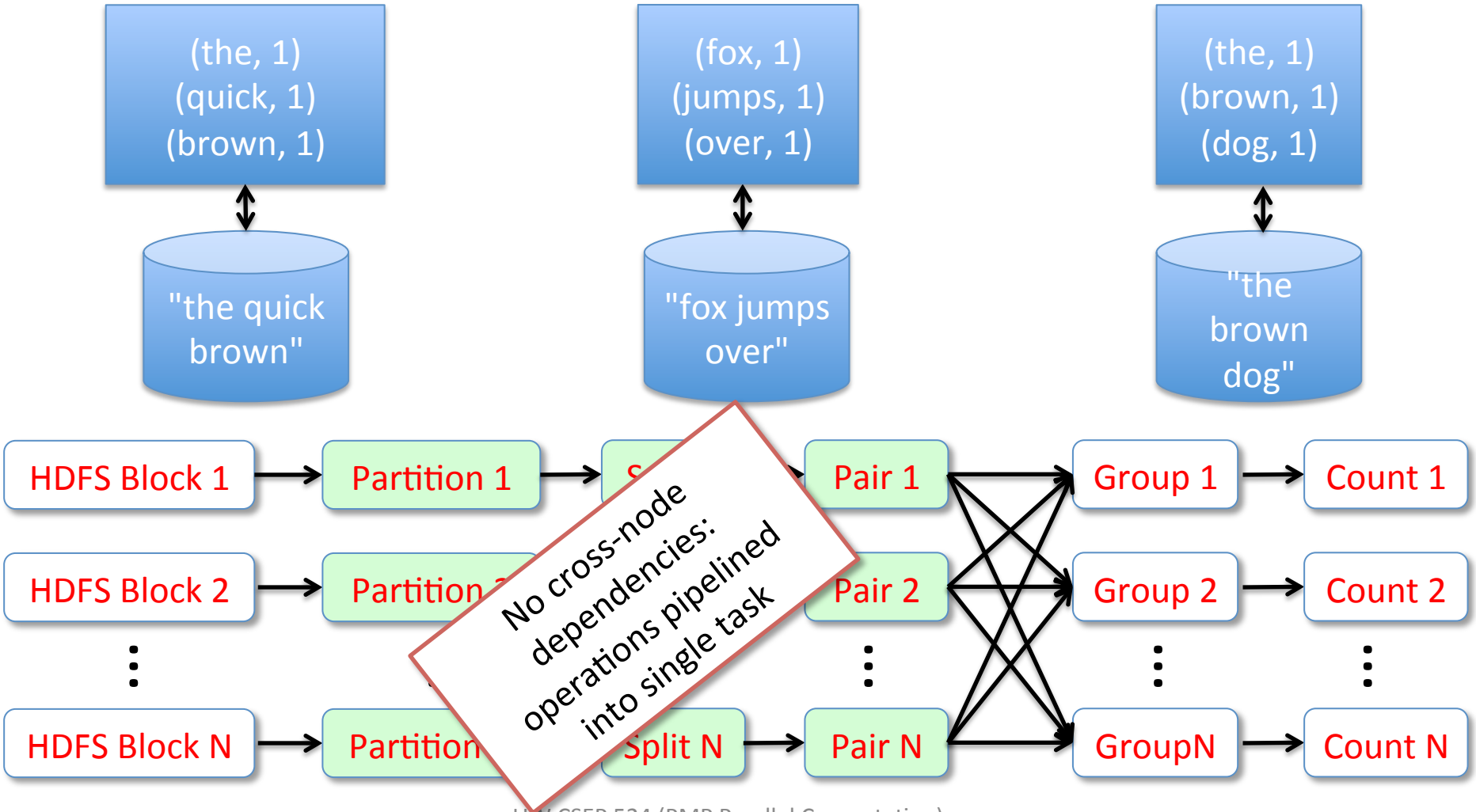

"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
HDFS Block N
Partition N
Split N
Pair N
GroupN
Count N

Execution


UNIVERSITY OF WASHINGTON
1861


(the, 1)
(quick, 1)
(brown, 1)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 1)
(brown, 1)
(dog, 1)
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
HDFS Block N
Partition N
Split N
Pair N
GroupN
Count N

Execution


(the, 1)
(quick, 1)
(brown, 1)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 1)
(brown, 1)
(dog, 1)
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
HDFS Block 2
HDFS Block N
Partition 1
Partition
Split N
Pair 1
Pair 2
Pair N
Group 1
Group 2
GroupN
Count 1
Count 2
Count N
No cross-node dependencies: operations pipelined into single task

Execution

(the, 1)
(quick, 1)
(brown, 1)

(fox, 1)
(jumps, 1)
(over, 1)

Barrier

(the, 1)
(brown, 1)
(dog, 1)

"the quick brown"

"fox jumps over"

"the brown dog"

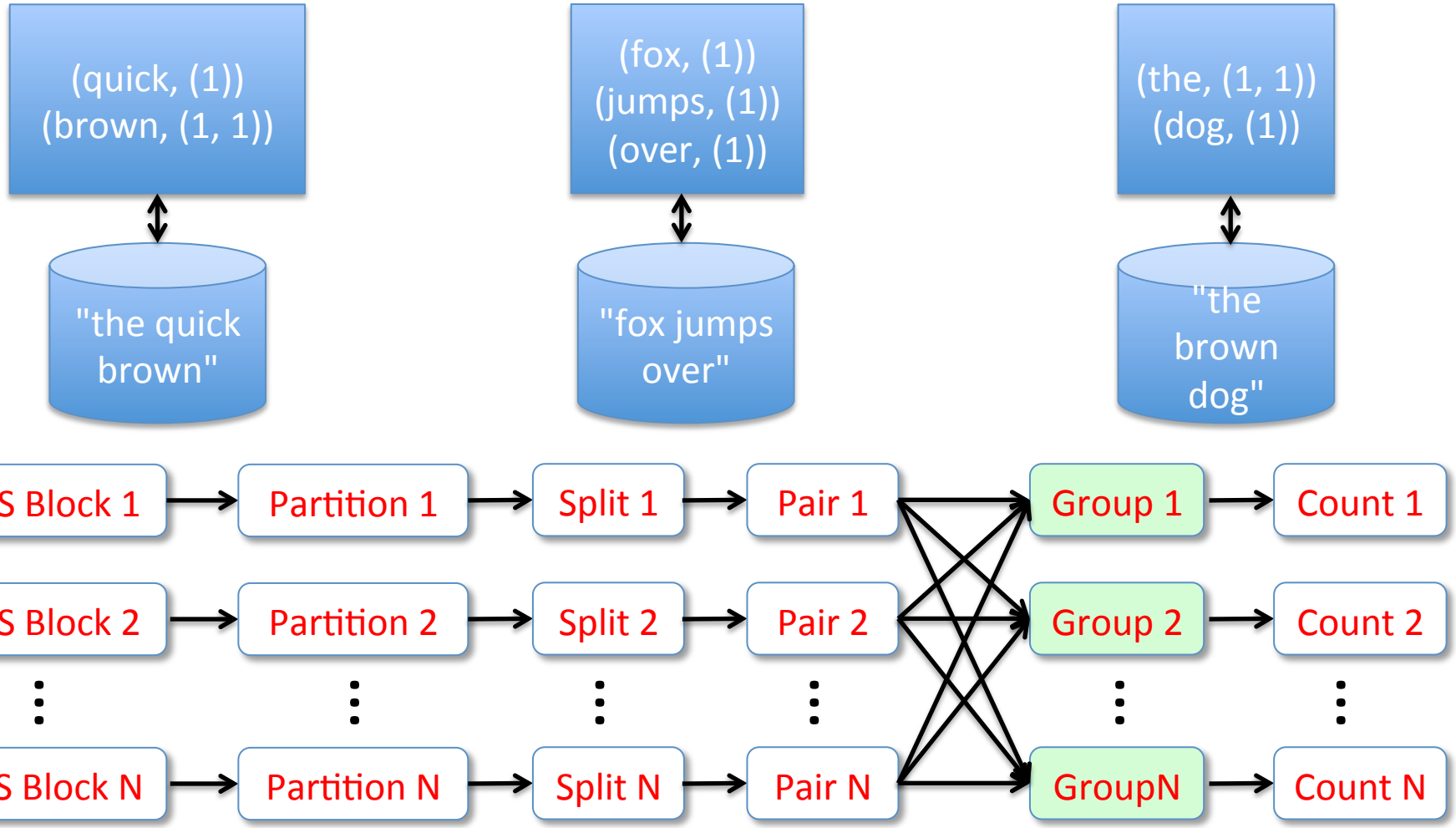
HDFS Block 1 → Partition 1 → Split 1 → Pair 1 → Group 1 → Count 1

HDFS Block 2 → Partition 2 → Split 2 → Pair 2 → Group 2 → Count 2

HDFS Block N → Partition N → Split N → Pair N → GroupN → Count N

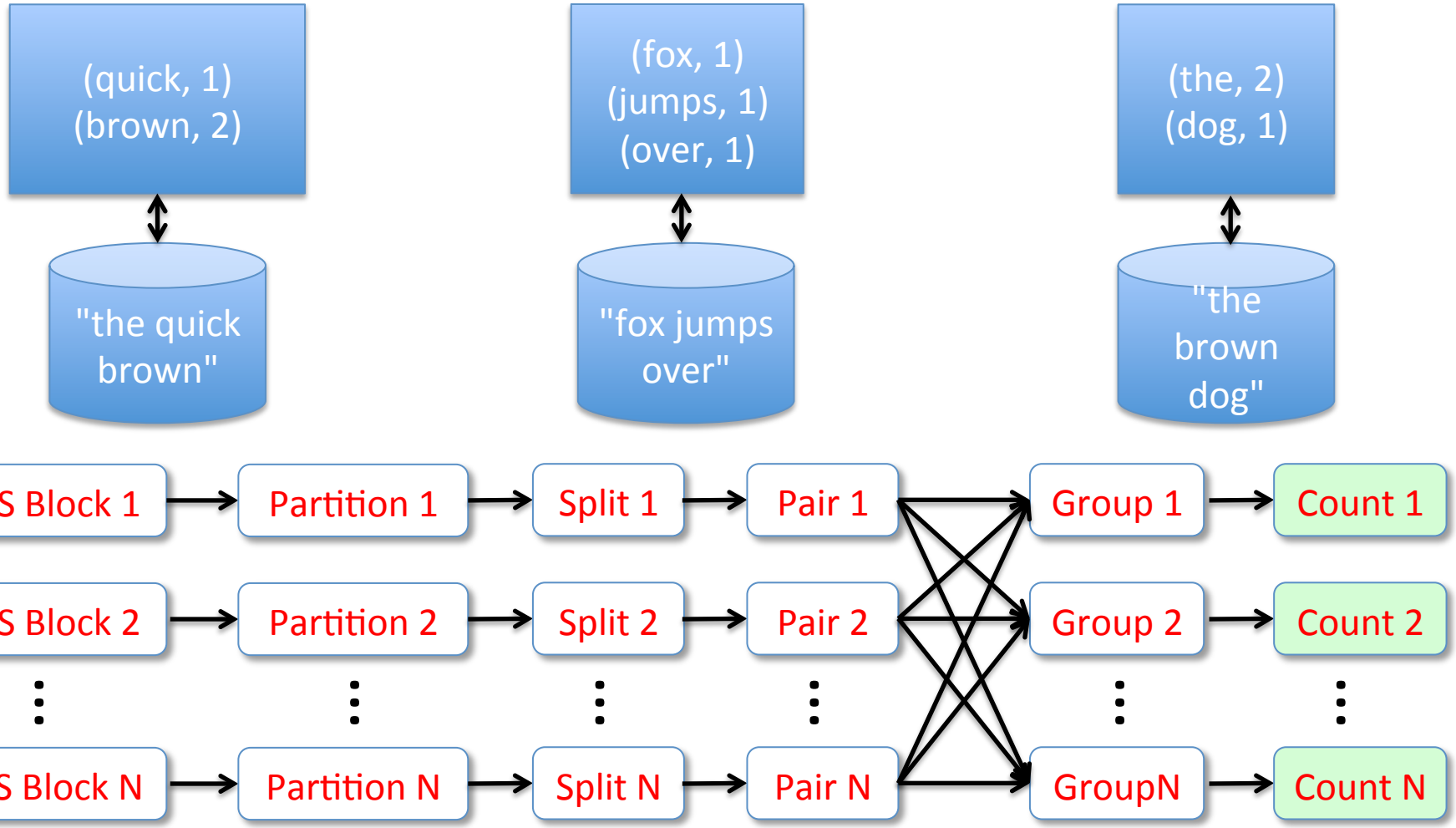
Execution


UNIVERSITY OF WASHINGTON
1861


(quick, (1))
(brown, (1, 1))
(fox, (1))
(jumps, (1))
(over, (1))
(the, (1, 1))
(dog, (1))
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
HDFS Block N
Partition N
Split N
Pair N
GroupN
Count N

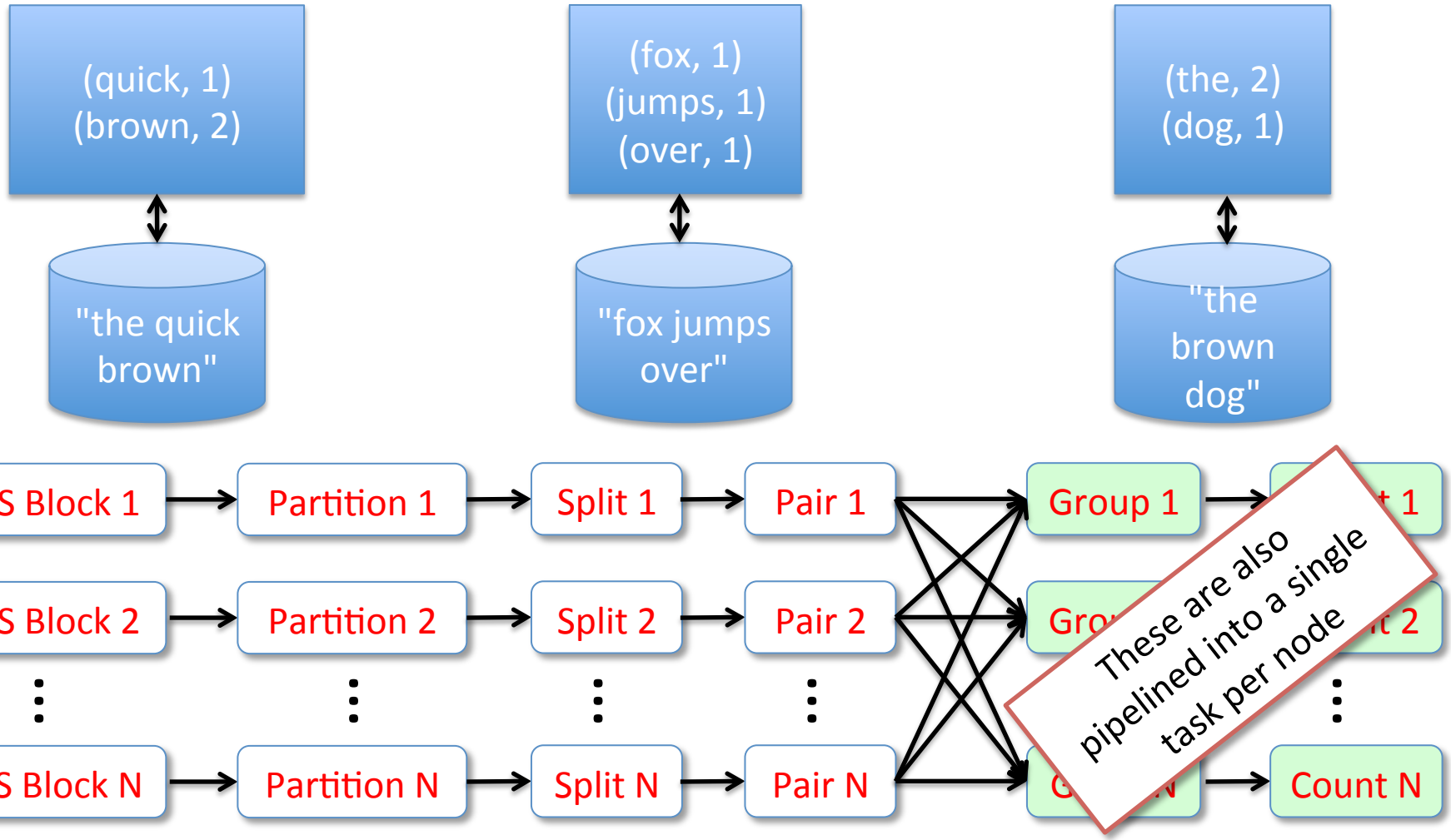
Execution


UNIVERSITY OF WASHINGTON
1861

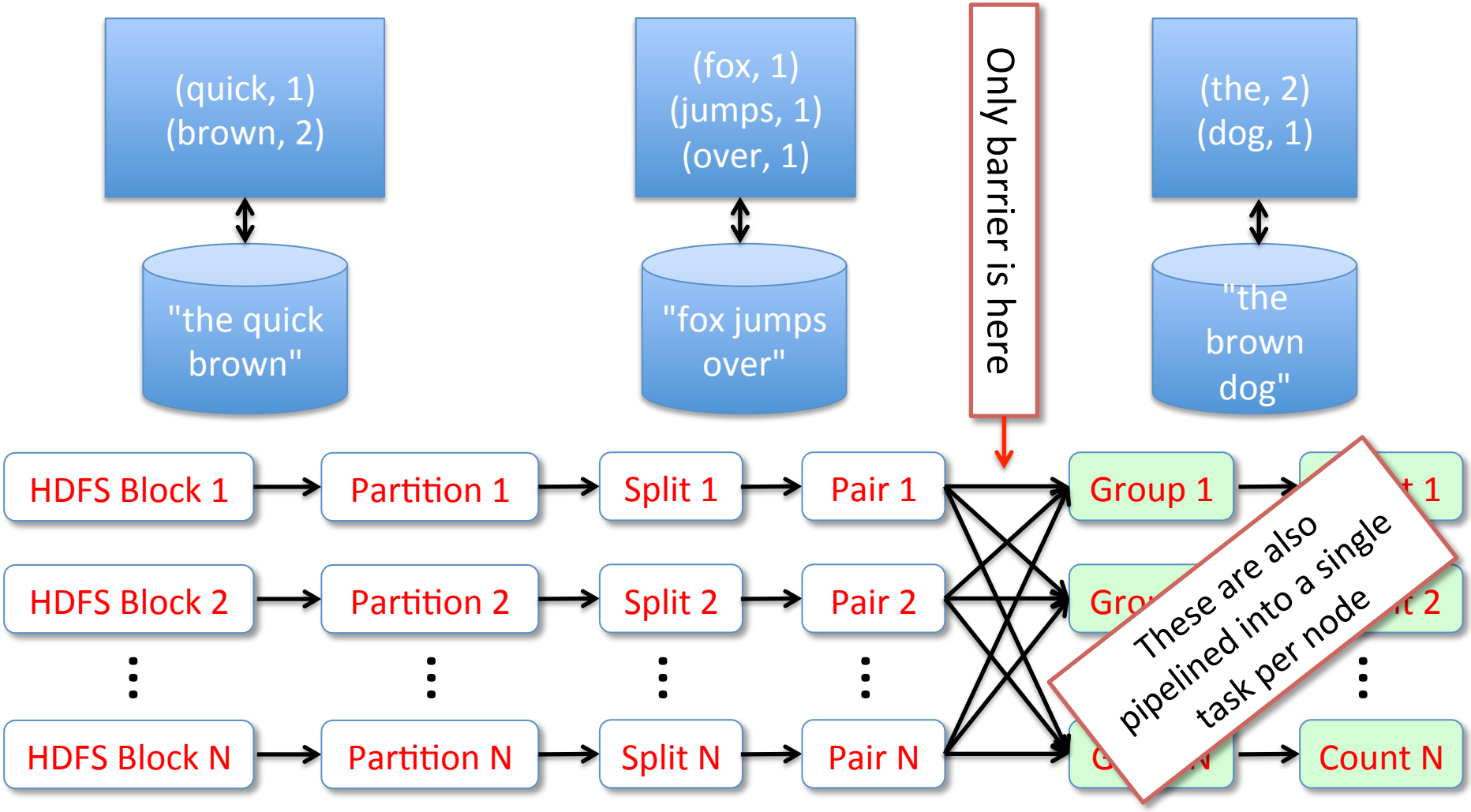

(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 2)
(dog, 1)
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
Count 1
HDFS Block 2
Partition 2
Split 2
Pair 2
Group 2
Count 2
HDFS Block N
Partition N
Split N
Pair N
GroupN
Count N

Execution


UNIVERSITY OF WASHINGTON
1861


(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
(the, 2)
(dog, 1)
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
HDFS Block 2
Partition 2
Split 2
Pair 2
HDFS Block N
Partition N
Split N
Pair N
Group 1
Count N
These are also pipelined into a single task per node

Execution

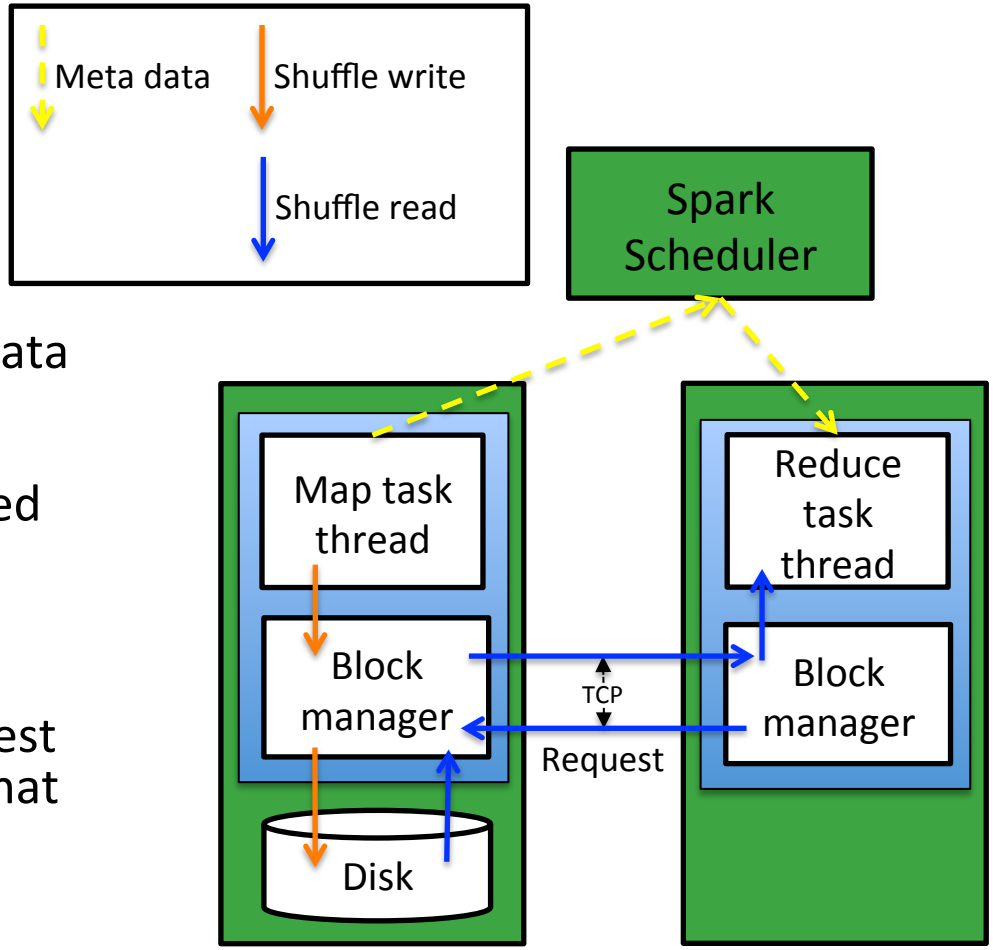

(quick, 1)
(brown, 2)
(fox, 1)
(jumps, 1)
(over, 1)
Only barrier is here
(the, 2)
(dog, 1)
"the quick brown"
"fox jumps over"
"the brown dog"
HDFS Block 1
Partition 1
Split 1
Pair 1
Group 1
HDFS Block 2
Partition 2
Split 2
Pair 2
HDFS Block N
Partition N
Split N
Pair N
Count N
These are also pipelined into a single task per node

Tasks and pipelining

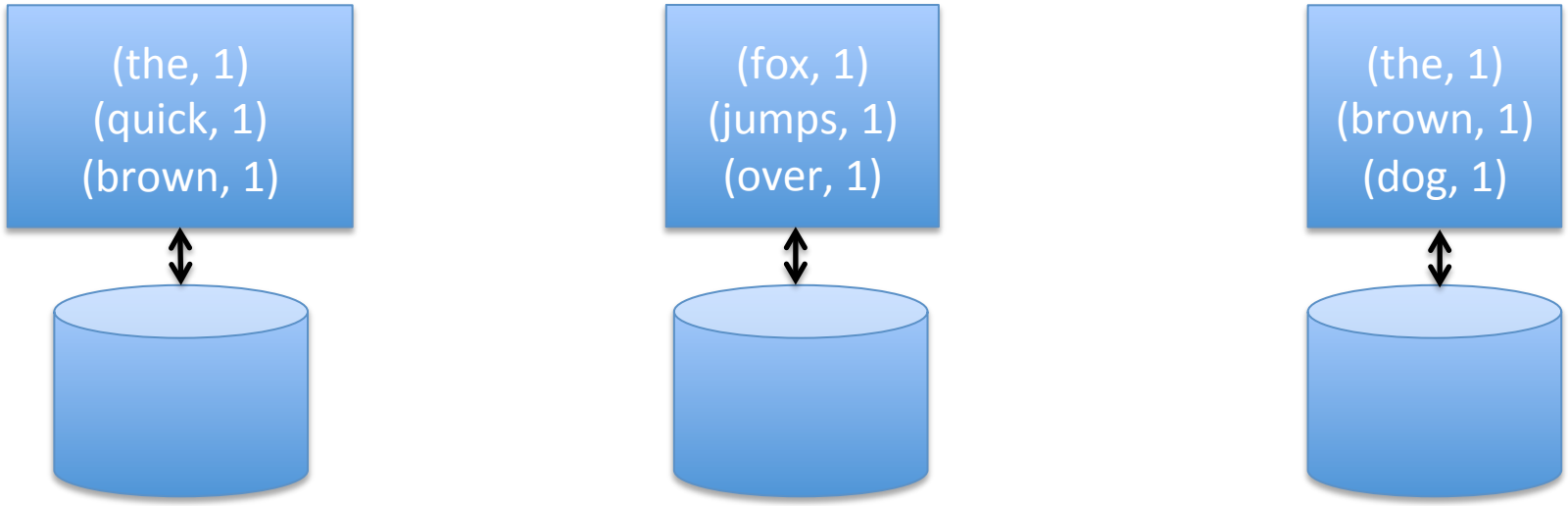
- If an RDD partitions's dependencies are on a single other RDD partition (or on *co-partitoned* data), the operations can be *pipelined* into a single *task*
  - **Co-partitioned**: all of the parent RDD partitions are co-located with child RDD partitions that need them
  - **Pipelined**: Operations can occur as soon as the local parent data is ready – no synchronization
  - **Task**: A pipelined set of operations
- Every task ends with a shuffle, and output or returning data back to the driver.

Shuffle implementation

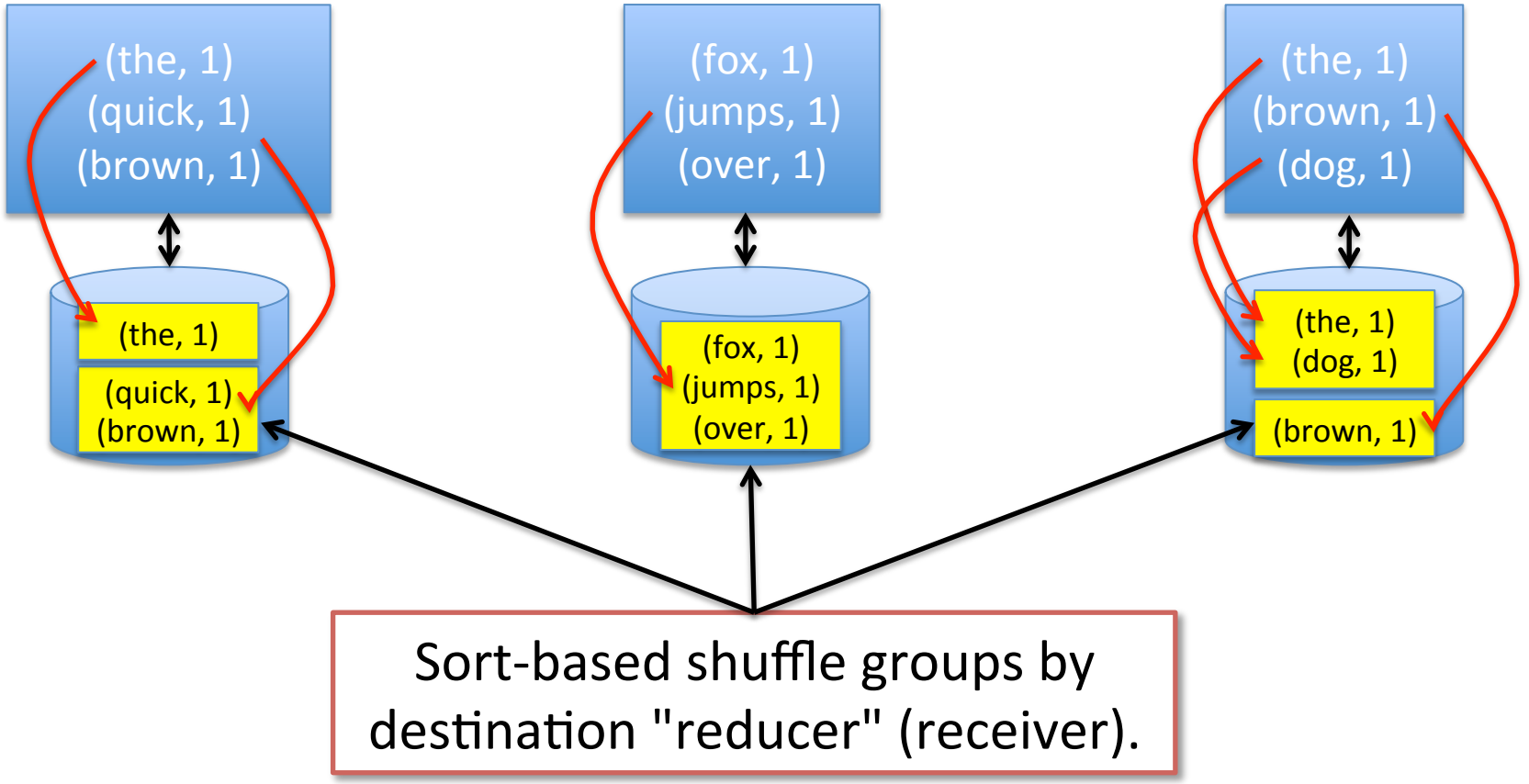
- All data exchanges between executors implemented via shuffle
  - Senders ("mappers") send data to block managers; block managers write to disks, tell scheduler how much destined for each reducer
  - Barrier until all mappers complete shuffle writes
  - Receivers ("reducers") request data from block managers that have data for them; block managers read and send



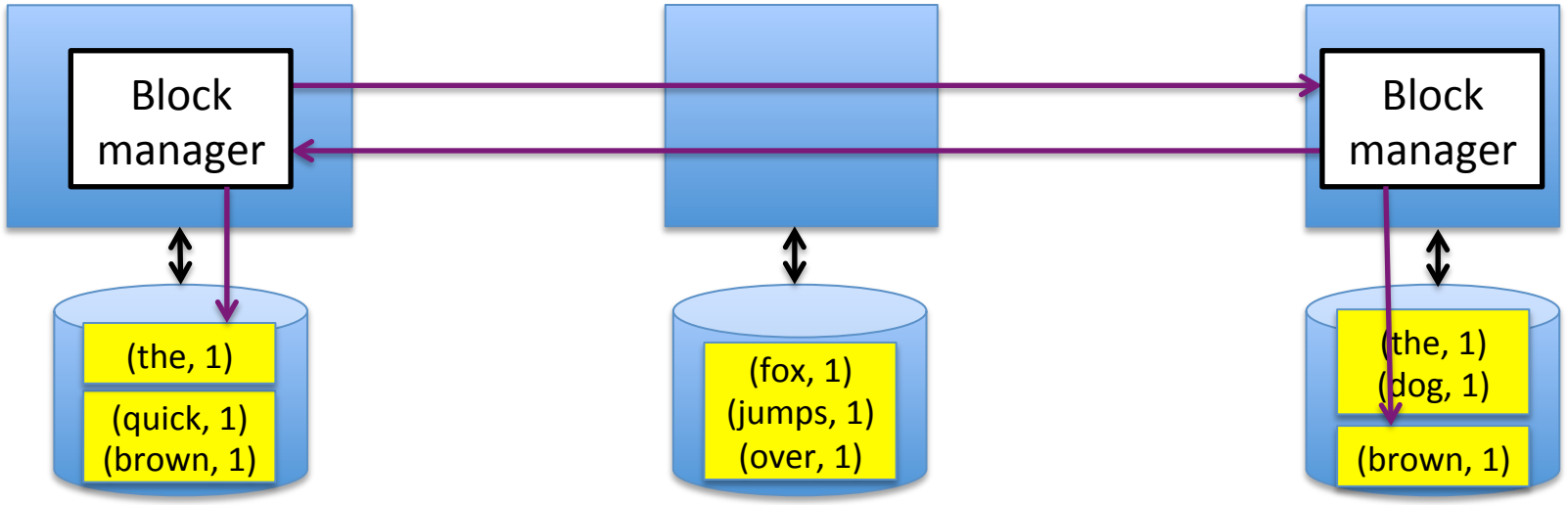
Shuffle Walkthrough



Shuffle Walkthrough

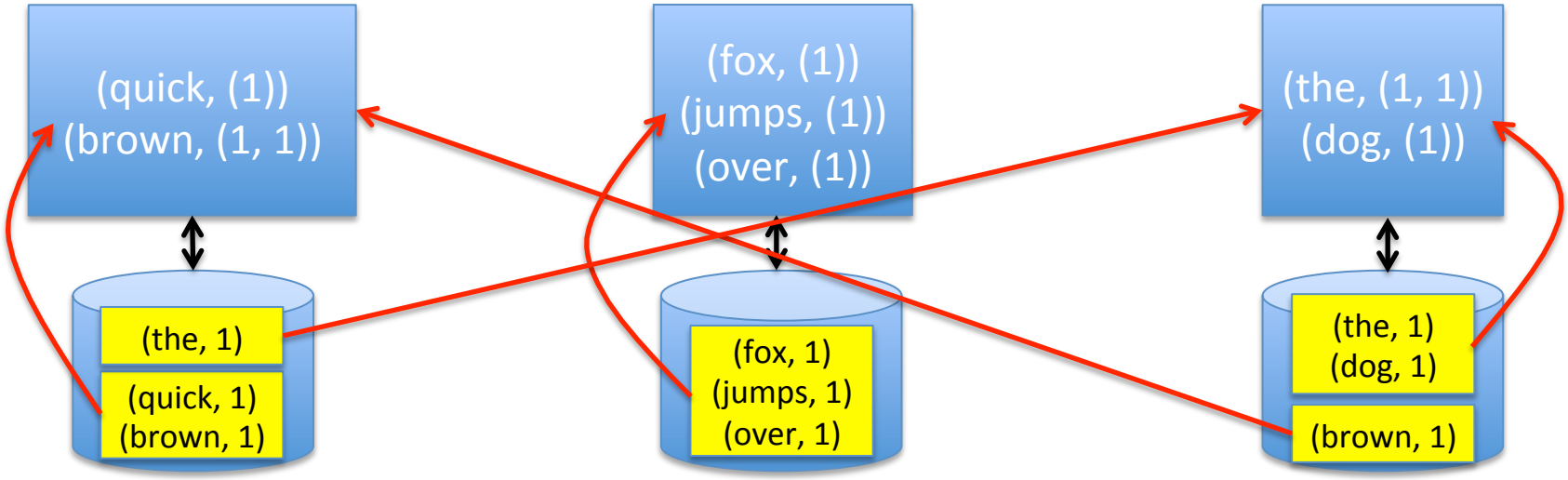


Shuffle Walkthrough



"Reducers" request blocks via block managers (only showing remote requests here to avoid clutter).

Shuffle Walkthrough



Data sent to reducers via block managers (BMs not shown).

Communication Example, Revisited

```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.reduceByKey(
                     (a,b) => a + b
                   )
val counts = wordCounts.collect()
```

- Can do this more efficiently with "reduceByKey" – aggregates results locally before shuffling
  - Like Schwartz's algorithm
  - Reduces amount of data sent over the network

Sample of Spark RDD Transformations

myRDD.map(function)	Generate new RDD by applying function to each element of myRDD
myRDD.filter(function)	Generate new RDD which contains elements of myRDD for which function returns true
myRDD.union(otherRDD)	Generate an RDD that contains all the elements of both myRDD and otherRDD
myRDD.intersection(otherRDD)	Generate an RDD that contains only elements which appear in both myRDD and otherRDD
myRDD.distinct()	Generate a new RDD that contains only the distinct elements of myRDD (duplicates removed)
myRDD.groupByKey()	If myRDD consists of key-value pairs, return a new RDD consisting of each key and a list of all of its values: e.g., (k1, x), (k2, y), (k1, z) → (k1, (x,z)), (k2, (y))

See http://spark.apache.org/docs/latest/programming-guide.html for more

Sample of Spark RDD Transformations

myRDD.reduceByKey(function)	Like groupByKey, but aggregate all values for each key using the parameter function
myRDD.join(otherRDD)	If myRDD and otherRDD are datasets of key-value pairs, where myRDD contains (K,V) pairs and otherRDD contains (K,W) pairs, generate a new RDD of (K, (V,W)) pairs for every V and W with the same key K.
myRDD.sortByKey()	Return a new RDD that contains the same key-value pairs as myRDD, but in sorted order (can optionally specify ascending or descending order).
myRDD.flatMap(function)	Generate a new RDD by applying function to each element of myRDD. Unlike map, function should return a sequence of 0 or more elements rather than a single element (so one original RDD element may generate 0 or many new RDD elements).

See http://spark.apache.org/docs/latest/programming-guide.html for more

Sample of Spark RDD Actions

myRDD.reduce(function)	Perform a reduction using function across all elements of an RDD, returning result to driver (e.g., sum all elements of the RDD).
myRDD.count()	Return the total number of elements in the RDD (closest thing to a no-op action, since this data is always computed – can be used to force evaluation).
myRDD.collect()	Return all elements of myRDD. Make sure they will fit in the driver's memory!
myRDD.takeOrdered(n)	Return the first n sorted elements of myRDD (may optionally provide a comparator).
myRDD.countByKey()	Return a hashmap of (K, integer) pairs, where the integer is the number of elements with key K.

See http://spark.apache.org/docs/latest/programming-guide.html for more

Open Discussion

- Last formal lecture of the course … what do you want to discuss/ask about?
- Hard stop at 8:30 for first few presentations.