CSEP 590 – Programming Systems
University of Washington

Lecture 4: Programming System for Distributed Data Analytics

Michael Ringenburg
Spring 2017

Course News

- Presentations
  - Thanks for submitting your topics
  - Presentations will be weeks 8 ,9, and 10
    - 6, 6, and 7, unless somebody wants to volunteer to go early
  - I'll be putting together the schedule this weekend. If you have a date preference, please send it to me by Friday.
- Announcement: No class on May 2
- Today: Specialized programming systems for Data Analytics
- Next week: Garbage collection

Big Data

- As of 2012, 2.5 exabytes of data created *every day*
- Storage capacity doubling every 40 months
- Massive amount of data now exist/are being generated
- How can we understand/process/utilize this amount of data?
- Does it open new possibilities? New paradigms? New challenges?

The ~~3~~ ~~4~~ 5 V's of Big Data

- Challenges of big data typically lie in one (or more) of the following *V*'s
  - **Volume**: very large amount of data
  - **Velocity**: data coming in very rapidly
  - **Variety**: many different types of data
- Two additional V's are sometimes added:
  - **Veracity**: Large variance in the quality of the data/ difficulty in determining quality
  - **Variability**: Inconsistency in the data

Big Data Examples

- **Large Hadron Collider**: 600 million particle collisions per second
- **Twitter**: 500 million tweets per day
- **Cybersecurity**: Analyzing network/file/other log data – potentially high velocity, high volume
- **Banking**: Credit card fraud detection
- **Real estate**: Windermere using 100 million GPS trackers to estimate commute times for new home buyers

Big Data Examples

- **Social Media**: 50 billion Facebook photos per day
- **Bioinformatics/medicine**: Massive amounts of genomic data to analyze; patient treatment outcomes, etc.
- **Financial Trading**: Analyzing stock/bond/option transactions; determining compliance with regulations; new trading algorithms
- **Medicare fraud detection**: analyzing medical billing records

Data Analytics

- Gleaning information from big data sets
  - Summarizing large data sets
  - Finding patterns
  - Looking for anomalies
  - Developing new models/theories
- Data scientists: experts in data analytics. Typically combine backgrounds in:
  - Statistics
  - Computer Science
  - Often domain-specific knowledge

Analytics: The Need for Parallel/ Distributed Computing

Big Data Analytics Challenge	Parallel Computing Solution
High volume of data	Many cluster nodes = large amount of memory to store data (DRAM and disks)
High velocity of data – need to keep up with rapid streams of new data	Many processor cores/nodes/threads to handle incoming data. Can scale up as velocity increases. Can dedicate some nodes solely to handling incoming data streams, leaving others for more complex processing.
Extracting knowledge from large quantities of data	Large datasets tend to scale up well to large numbers of processes. Parallel versions of many common machine learning/statistical algorithms are well studied
Veracity and Variability	Data cleaning and preparation tasks often parallelize well.

Productivity-oriented Analytics Frameworks

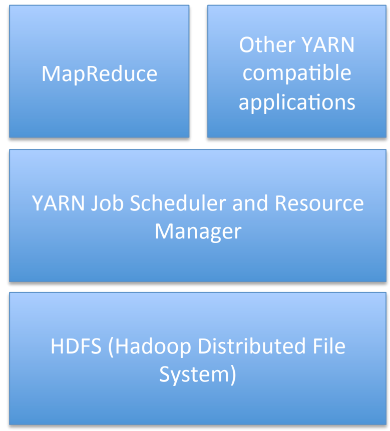
- Data Scientists want to quickly explore data, find patterns, etc …
  - Not fight to parallelize their code and efficiently distribute their data
  - May have some CS training, but not always, and often not specialized
- Ideally, want a programming system that:
  - Supports high level language(s?) with extensive library support (e.g., Python, R, maybe Java or Scala)
  - Supports common analytics tasks: SQL, statistics, machine learning algorithms, graph algorithms, etc
    - Either built-in or easily extensible
  - Simplifies parallelism and distribution of data
  - Simplifies management of cluster of workers

Analytics: Hardware Environments

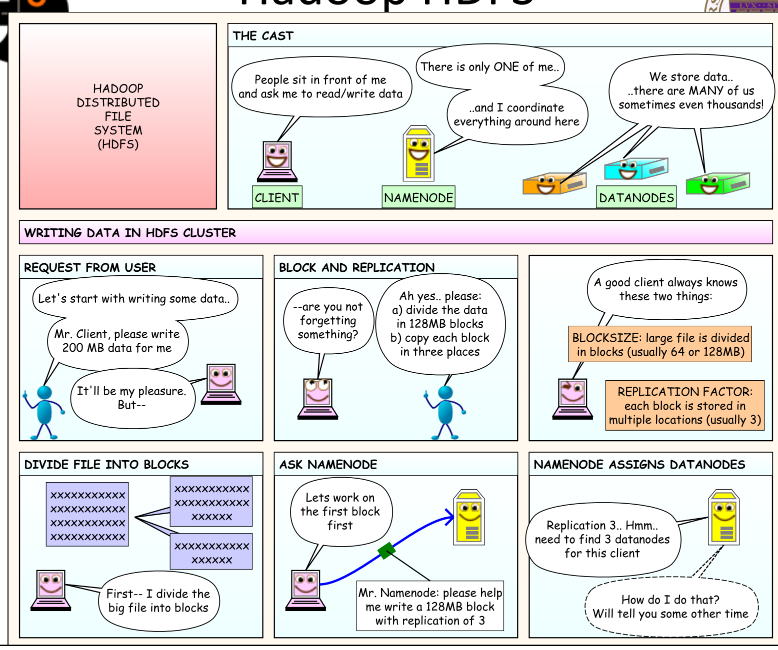
- Typical analytics framework assumes:
  - Loosely connected cluster (e.g., 1 Gigabit Ethernet)
  - Cheap/unreliable hardware
  - Often, many "spindles" per node (local hard drives) – i.e., high bandwidth to local storage, but also high latency. Seeing more and more SSD-based solutions, however, in higher-end market.
- Frameworks optimized, configured, designed for this environment.

Hadoop Framework

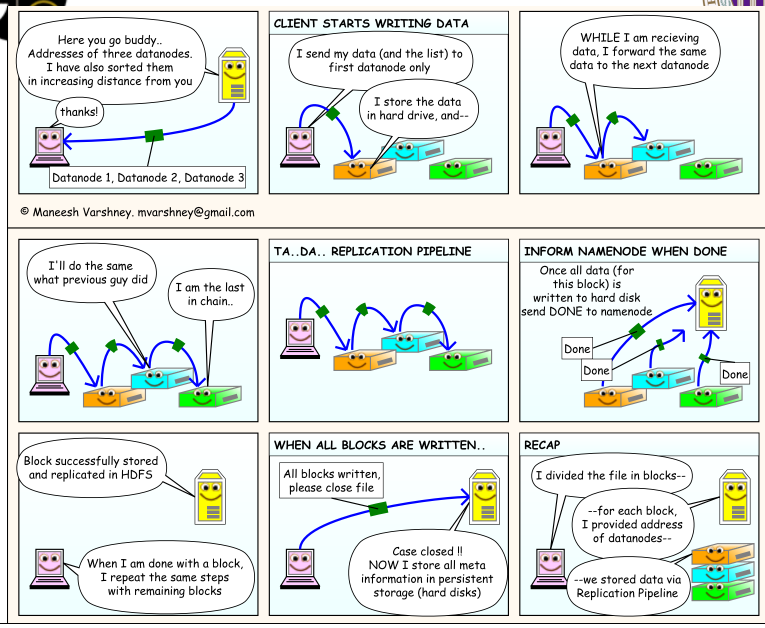
- HDFS: distributed file system, uses replication for fault tolerance, supports locality ("move compute to data")
  - **Simplify data distribution**
- YARN: assigns cluster resources (e.g., memory, cores) to jobs, schedules execution
  - **Simplify cluster management**
- MapReduce: Application framework – maps and reduces, based on Jeff Dean paper linked on course web
  - **Simplify parallel programming**
- Can run other frameworks on top of Yarn (e.g., Spark, Giraph, Hive)
  - **Extensibility**



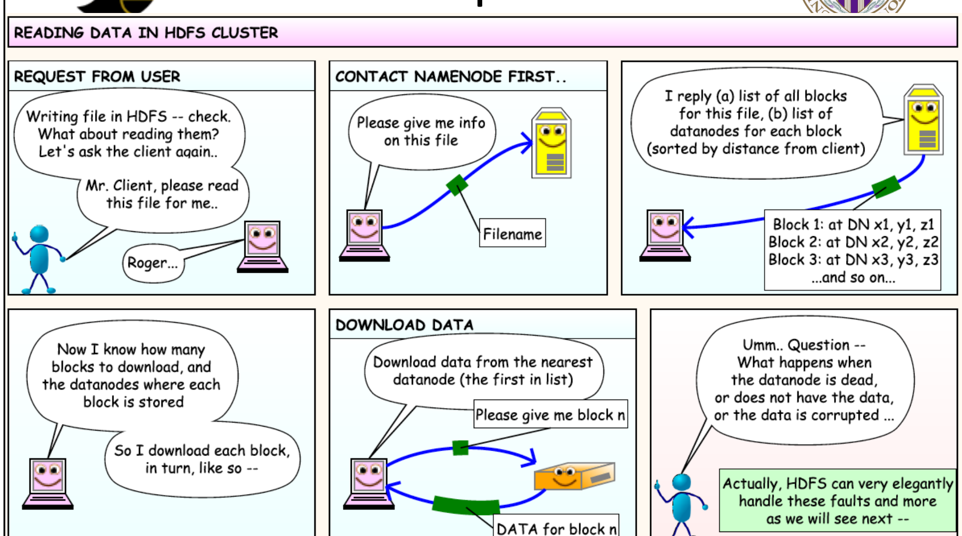
Hadoop HDFS



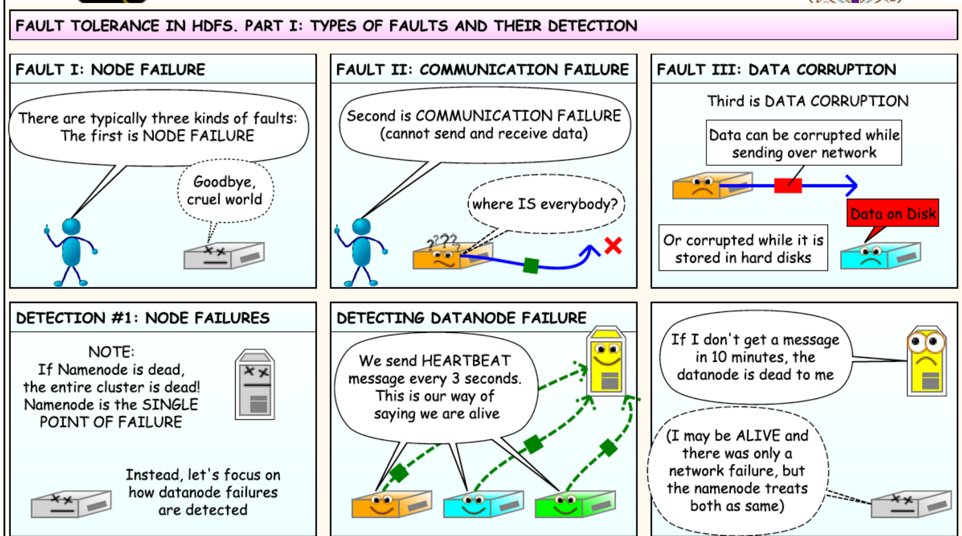
Hadoop HDFS



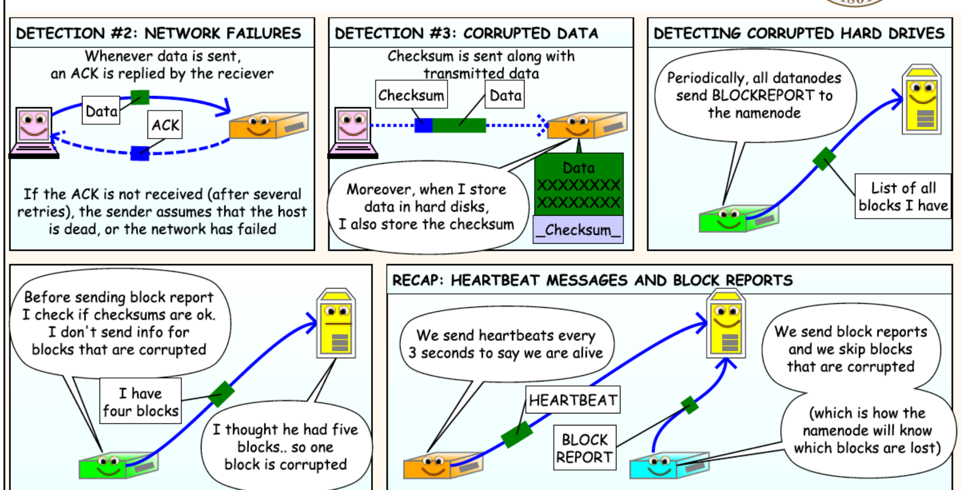
Hadoop HDFS



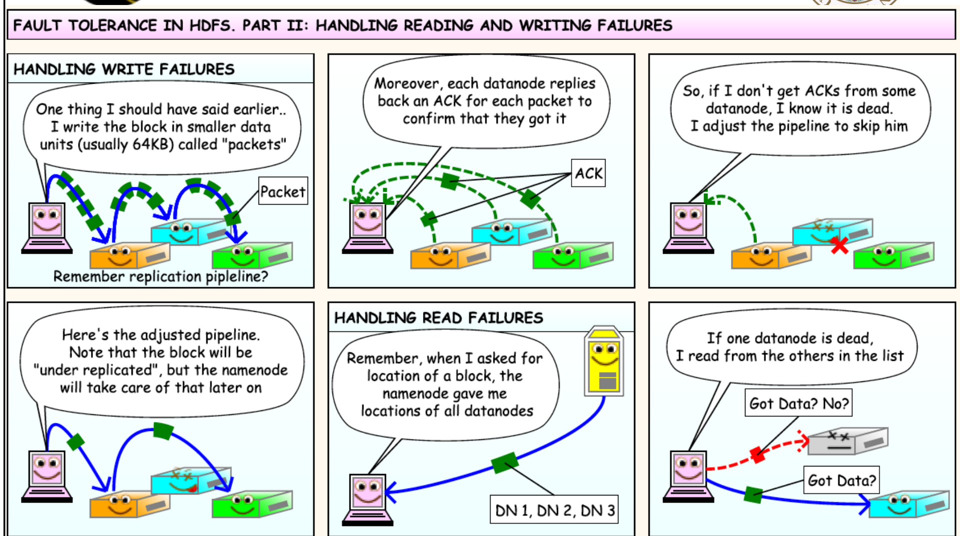
Hadoop HDFS



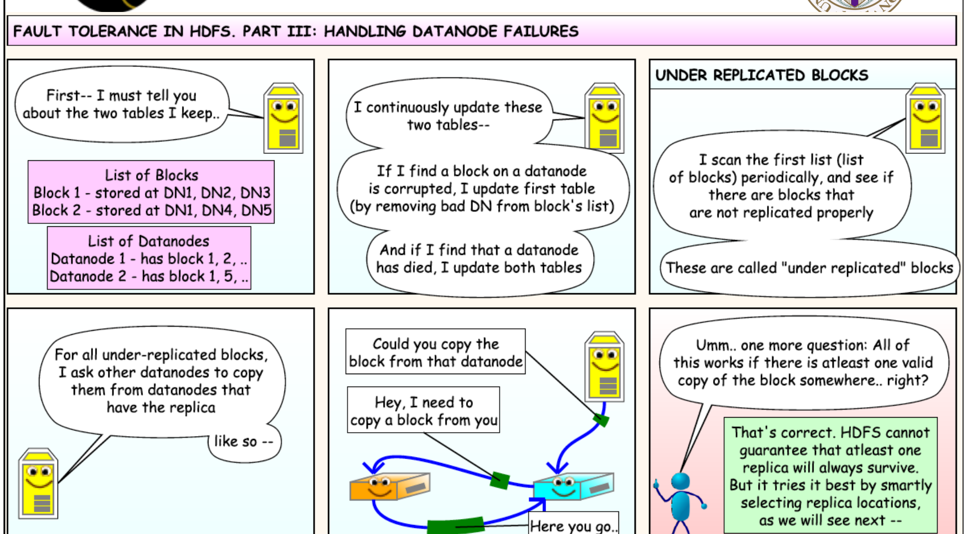
Hadoop HDFS



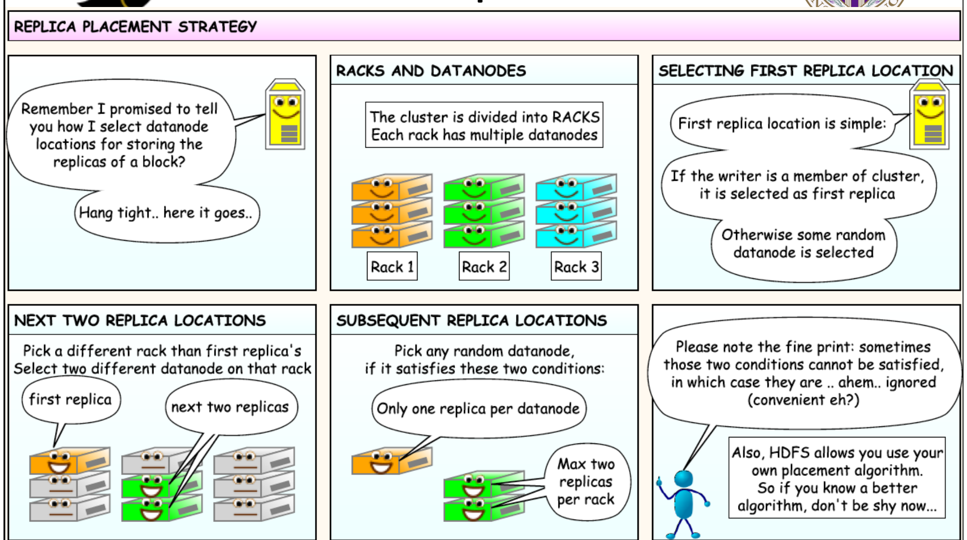
Hadoop HDFS



Hadoop HDFS



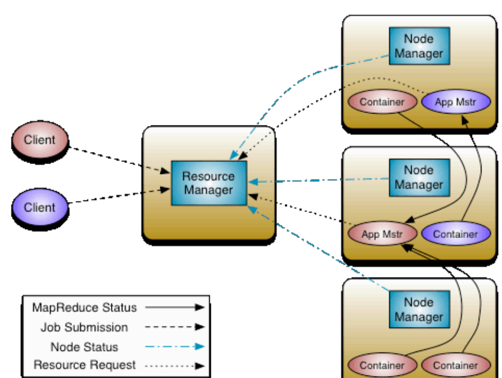
Hadoop HDFS



- Cartoon ©Maneesh Varshney (https://drive.google.com/file/d/0B-zw6KHOtbT4MmRkZWJjYzEtYjI3Ni00NTFjLWE0OGItYTU5OGMxYjc0N2M1/view)

Hadoop Yarn

- **Resource Manager**: Global engine responsible for arbitrating job resource requests, determining when jobs run
- **Containers**: Job processes run inside *containers* – essentially the resources allocated to the job on each node
- **Node Manager**: Per-machine, launches application containers, ensures they don't exceed resource allocations
- **Application master**: Runs in first container, requests resources for rest of job, manages job



From hadoop.apache.org

Hadoop MapReduce

- You saw this concept in the Jeff Dean paper
- Mappers (implement `Mapper` interface) process (ideally local) key-value pairs, convert ("map") them to new key-value pairs.
- Data shuffled and sorted, so that all pairs with same key land on same node
- One reducer process per key (implements `Reducer` interface) processes/summarizes all data with the same key

MapReduce Word Count

```
public static class Map extends MapReduceBase implements
  Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value,
                  OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
```

MapReduce Word Count

```
public static class Reduce extends MapReduceBase implements
  Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text, IntWritable> output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```


Combiners

- Optional class – performs local reductions
  - Mappers collect key-value pairs in lists: one per key
  - Combiner method applied to each list prior to sending to reducer
  - When combiner buffer full, flushed by sending to reducer
  - Reduces communication
  - Word Count example – combiner adds counts:
    - (the, 1), (the, 1), (and,1), (the,1) -> (the,3), (and,1)

MapReduce: More Complex Algorithms?

- More complex algorithms often require multiple passes of MapReduce
  - Each pass generates key-value pairs
  - Next pass uses key-value pairs from previous pass as input
- E.g., KMeans clustering:
  - Map:
    - Read cluster centers from disk
    - Compute nearest center to each data point, put it in that cluster
    - Write data points (values) in each cluster (key)
  - Reduce
    - Combines all points in each new cluster, compute average
    - This is new cluster center – write it
  - Repeat until no more changes
- New frameworks like Spark are more flexible/don't require such contortions

Hadoop Performance Issues

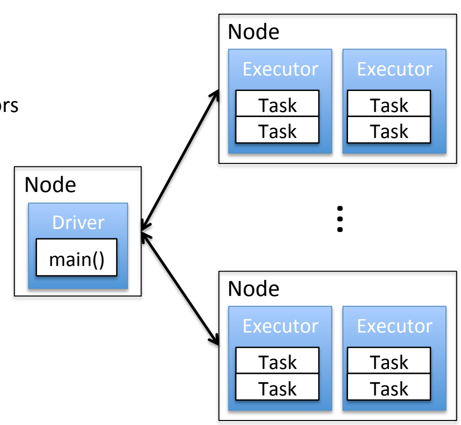
- Conventional wisdom: gated on IO bandwidth, network interconnect bandwidth
  - Shuffle: Writing mapper output to disk, sending over network, reading reducer input form disk
  - Complex algorithms often require multiple phases of map-reduce – HDFS I/O between each
  - Rule of thumb: "spindle-per-core" (or per 2 cores for compute intensive jobs)
- Intermediate data lost – often have to regenerate during next map-reduce

Apache Spark

- Tries to address two of key performance issues of Hadoop:
  - Allows "persisting" intermediate data
  - Can pipeline many operations in a single stage, keeps in memory except when shuffle necessary (or spill if runs out of memory)
- Often nice performance gains
- Also, programming flexibility – not constrained to rigid Map then Reduce paradigm (but often ends up similar)

Spark Execution Model

- "Master-slave" parallelism model
- Driver (master)
  - Executes main
  - Distributes data and work to executors
- Resilient Distributed Dataset (RDD)
  - Spark's ~~primary~~ original data abstraction
  - Partitioned amongst executors
  - Fault-tolerant via lineage
  - *Newer data abstractions like dataframes and dataset follow the same basic model*
- Executors (workers)
  - Lazily execute tasks (operations on partitions of the RDD)
  - Global all-to-all shuffle (with barrier) for data exchange



RDD In Depth

- Original data abstraction of Spark
  - DataFrames adds rows and named columns (originally SchemaRDD)
  - DataSets add strong typing to DataFrames
- 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)
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 32

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)
```

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 33

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 …

Driver:
{1, …, 1,000,000}

Executor 0: | Executor 1: | Executor 2: | Executor 3:

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

Example: Line-by-line

Conceptually …

Driver:
{1, …, 1,000,000}

Executor 0:
{1 … 125000}
{500001 … 625000}

Executor 1:
{125001 … 250000}
{625001 … 750000}

Executor 2:
{250001 … 375000}
(750001 … 875000}

Executor 3:
{375001 … 500000}
(875001…1000000}

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


Example: Line-by-line

Conceptually …

Driver:
{1, …, 1,000,000}

Executor 0:
{2,4, … 125000}
{500002,500004 …}

Executor 1:
{125002, 125004 …}
{625002, 625004 …}

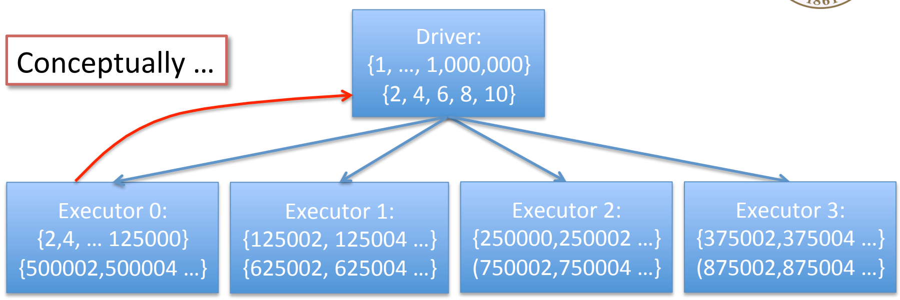
Executor 2:
{250000,250002 …}
(750002,750004 …}

Executor 3:
{375002,375004 …}
(875002,875004 …}

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

Example: Line-by-line

Conceptually …




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


Example: Line-by-line

Now let's try it out …

What's going on here?

Lazy Evaluation:

Driver:
{1, …, 1,000,000}

Executor 0:

Executor 1:

Executor 2:

Executor 3:

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


What's going on here?

Lazy Evaluation:

Driver:
{1, …, 1,000,000}

Executor 0:

Executor 1:

Executor 2:

Executor 3:

Input: Arr1M

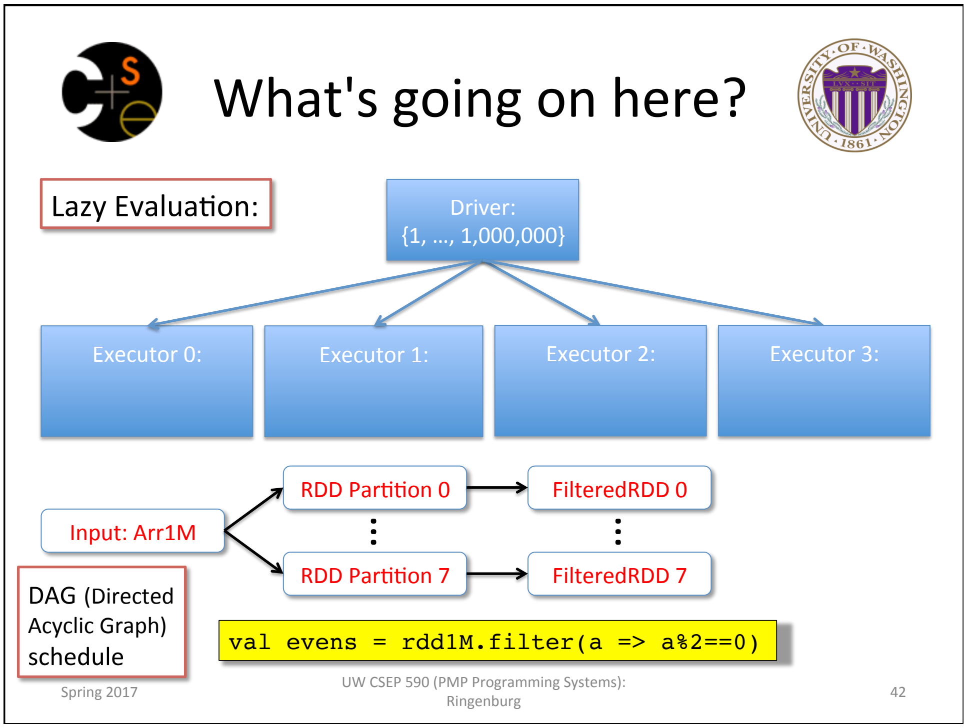
RDD Partition 0

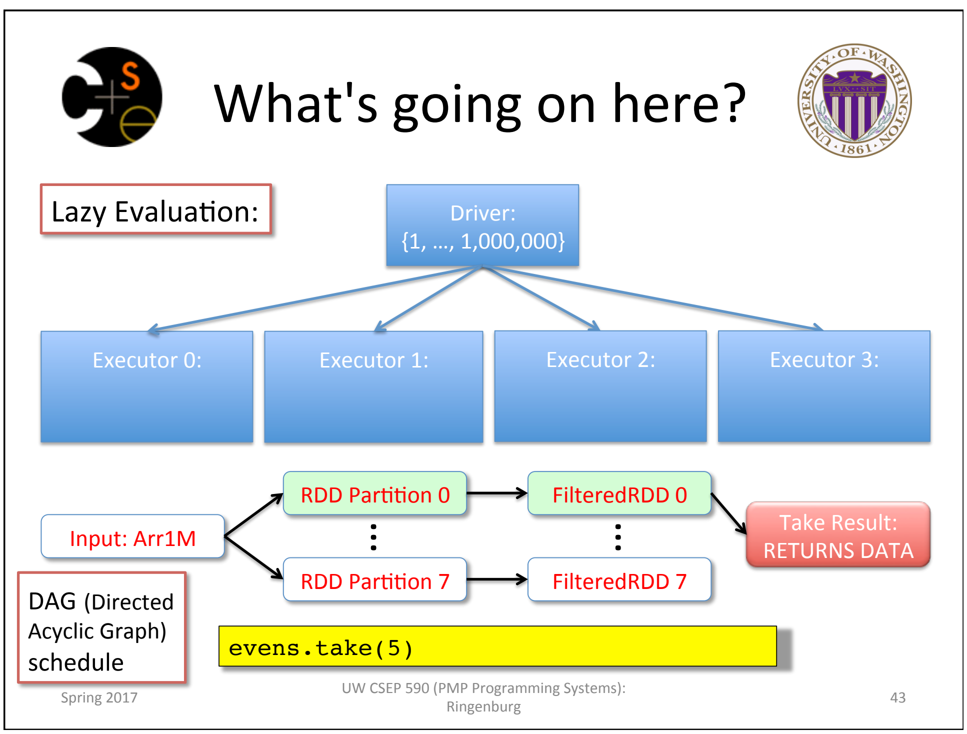
⋮

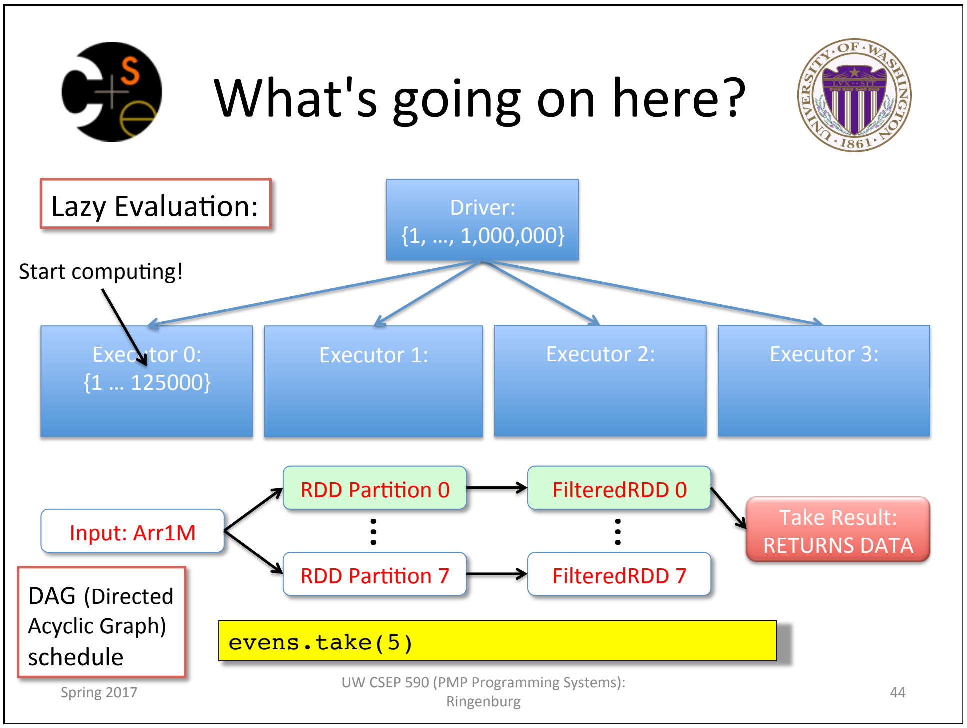
RDD Partition 7

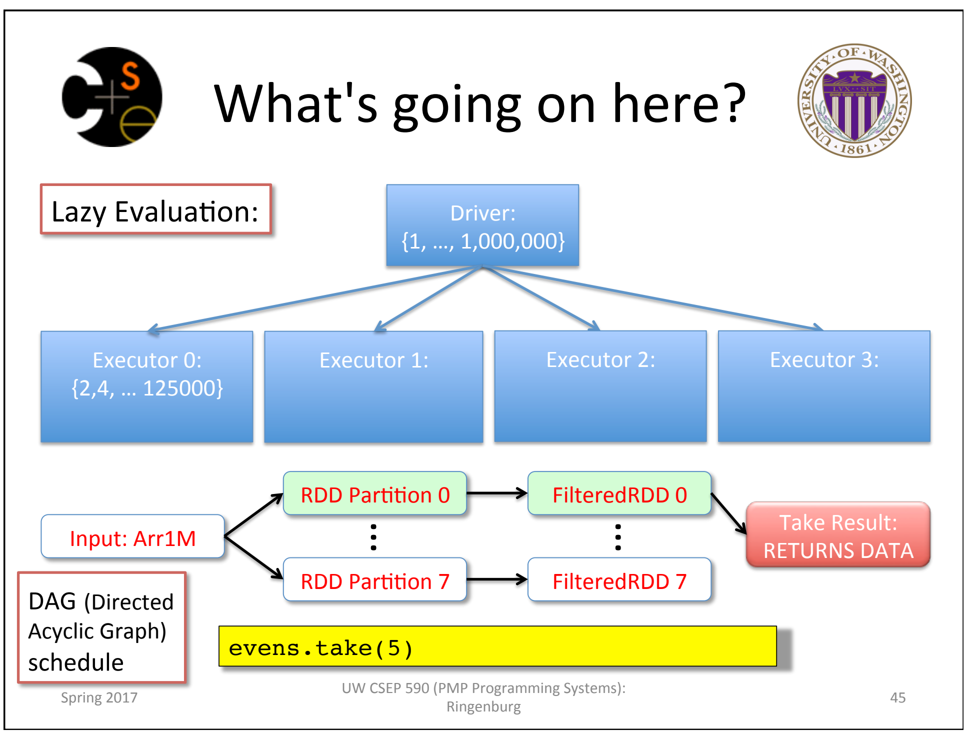
DAG (Directed Acyclic Graph) schedule

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

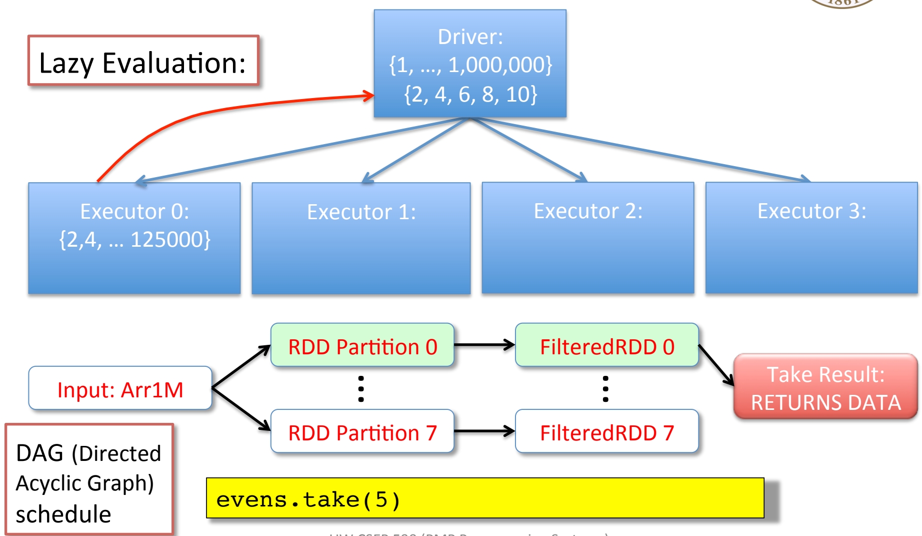

What's going on here?
Lazy Evaluation:
Driver:
{1, …, 1,000,000}
Executor 0:
Executor 1:
Executor 2:
Executor 3:
Input: Arr1M
RDD Partition 0
RDD Partition 7
FilteredRDD 0
FilteredRDD 7
DAG (Directed Acyclic Graph) schedule
val evens = rdd1M.filter(a => a%2==0)
Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
42


What's going on here?
Lazy Evaluation:
Driver:
{1, …, 1,000,000}
Executor 0:
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)
Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
43


What's going on here?
Lazy Evaluation:
Driver:
{1, ..., 1,000,000}
Start computing!
Executor 0:
{1 ... 125000}
Executor 1:
Executor 2:
Executor 3:
Input: Arr1M
RDD Partition 0
FilteredRDD 0
RDD Partition 7
FilteredRDD 7
Take Result:
RETURNS DATA
DAG (Directed Acyclic Graph) schedule
evens.take(5)
Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
44


What's going on here?
Lazy Evaluation:
Driver:
{1, ..., 1,000,000}
Executor 0:
{2,4, ... 125000}
Executor 1:
Executor 2:
Executor 3:
Input: Arr1M
RDD Partition 0
FilteredRDD 0
RDD Partition 7
FilteredRDD 7
Take Result:
RETURNS DATA
DAG (Directed Acyclic Graph) schedule
evens.take(5)
Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
45

What's going on here?



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

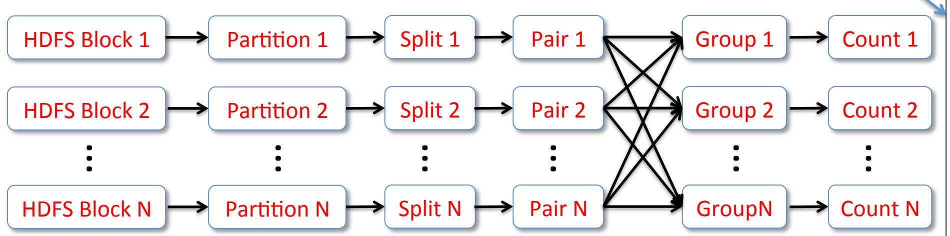
```
val lines = sc.textFile("mytext")
val words = lines.flatMap (
              line => line.split(" ")
            )
val wordKV = words.map(s => (s, 1))
val groupedWords = wordKV.groupByKey()
val wordCounts = groupedWords.map(
                   t => (t._1, t._2.sum)
                 )
val counts =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

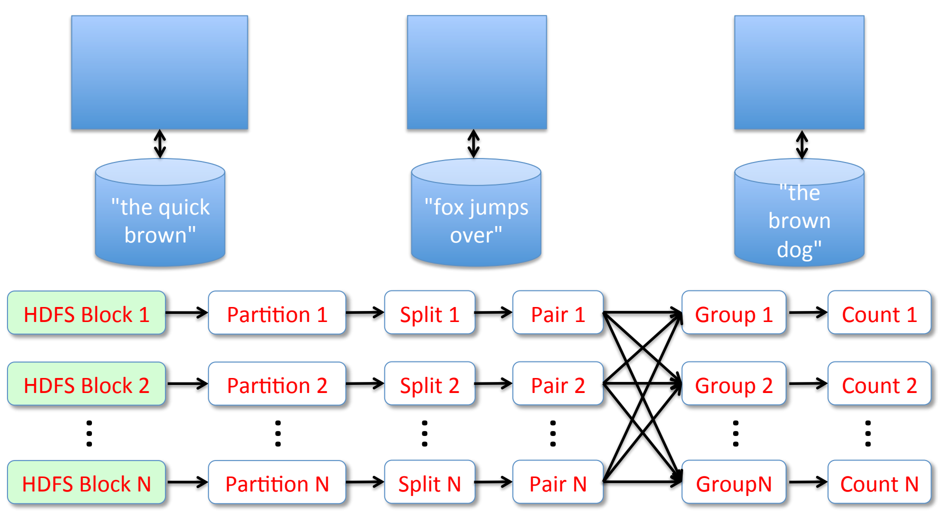
The DAG

```
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()
```

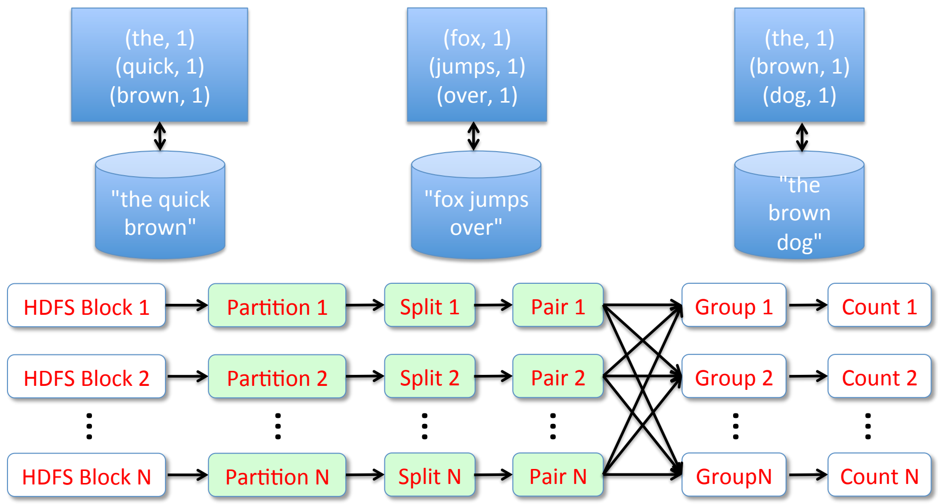
Omitting collect due to space constraints



Execution

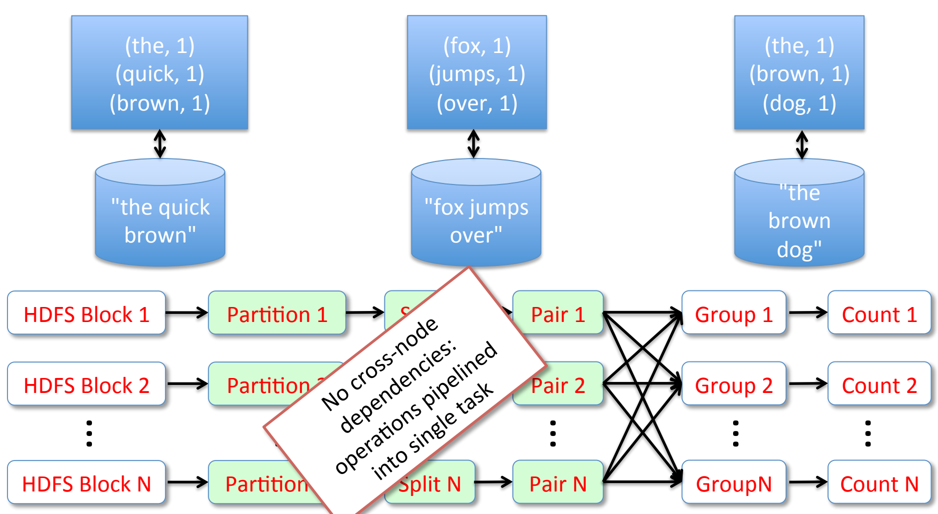


Execution



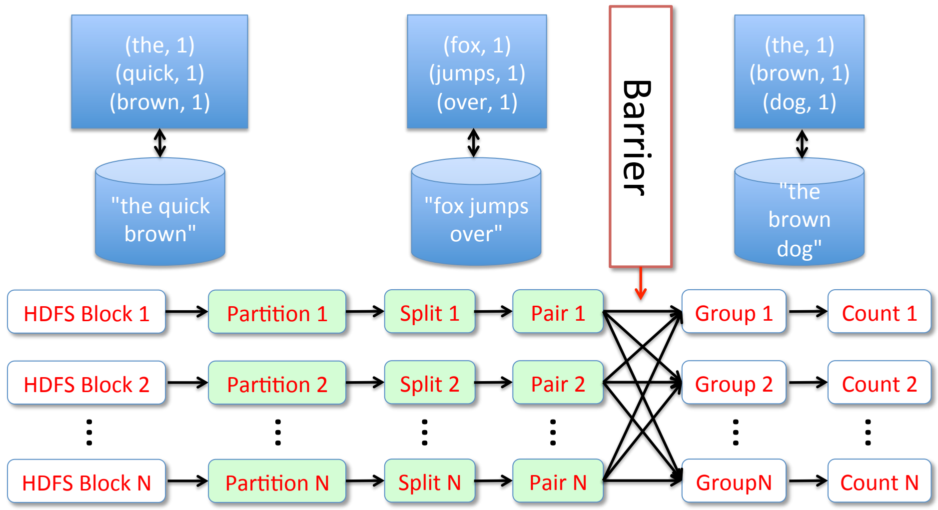
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 60

Execution



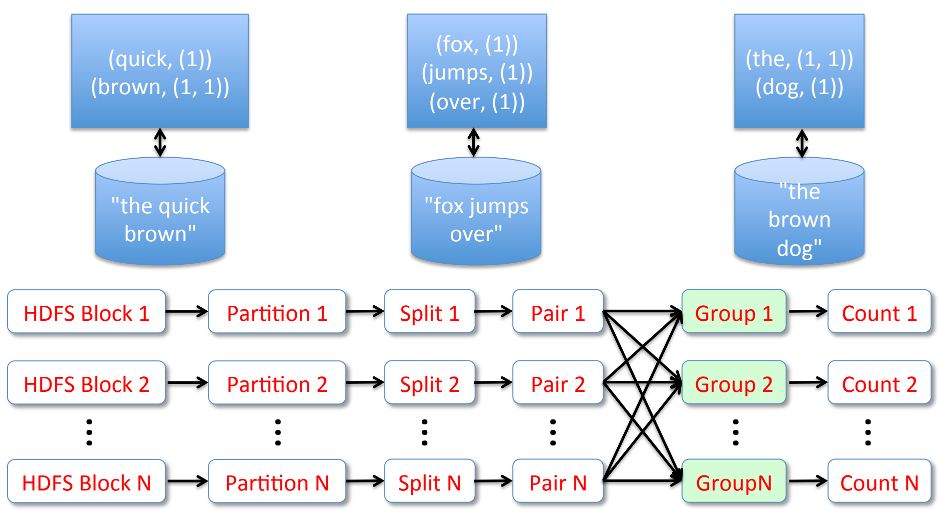
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 61

Execution



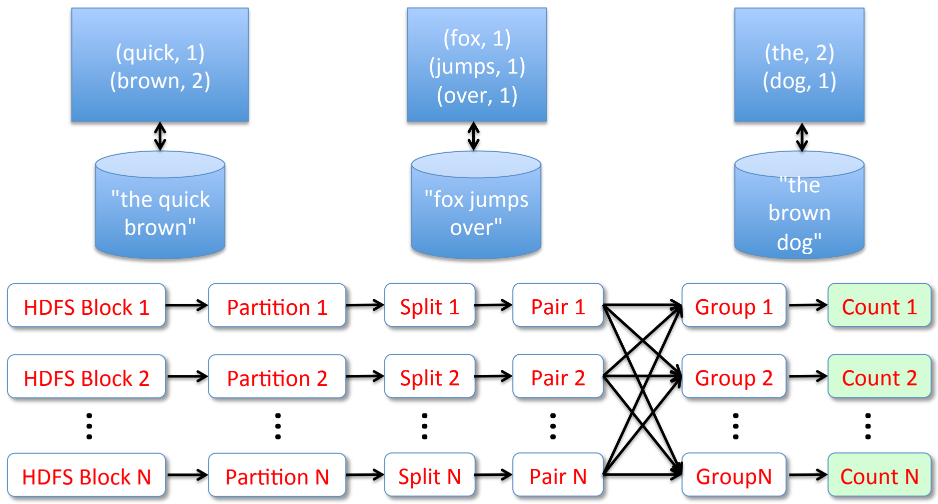
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 62

Execution



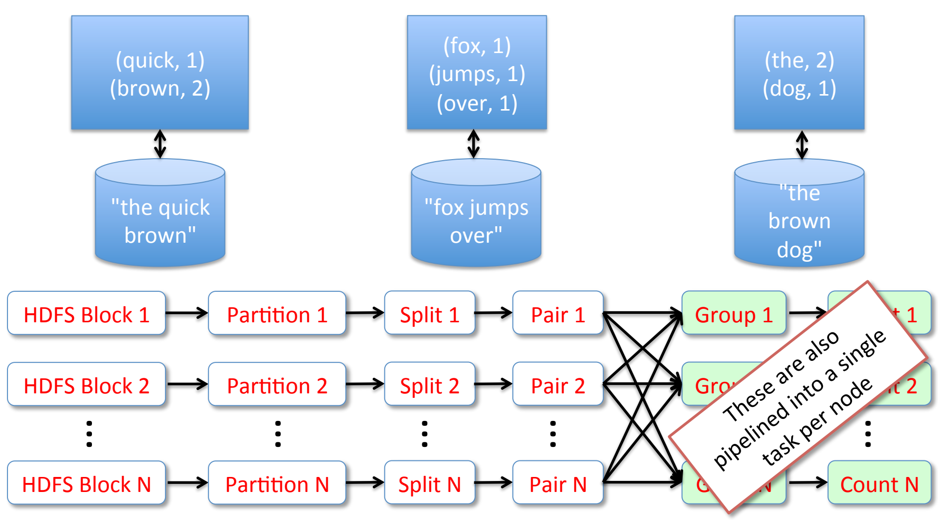
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 63

Execution



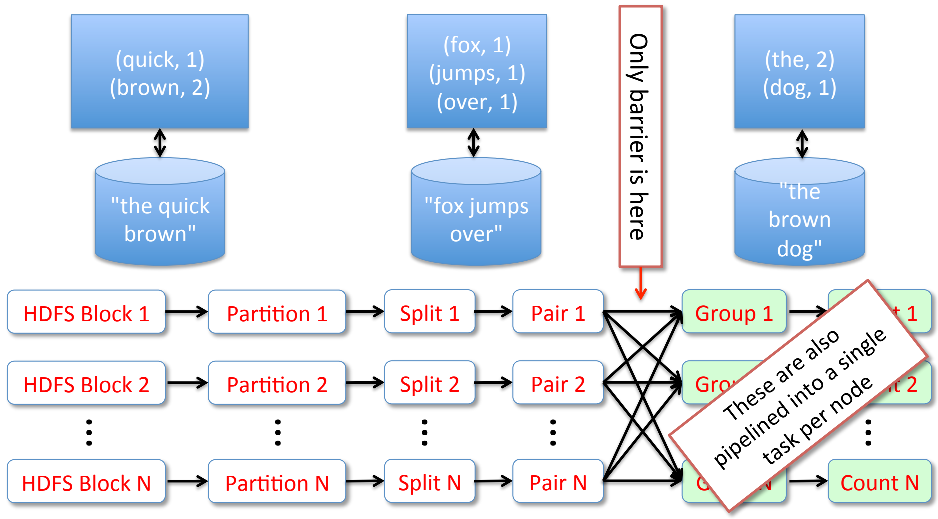
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 64

Execution



Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 65

Execution



Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 66

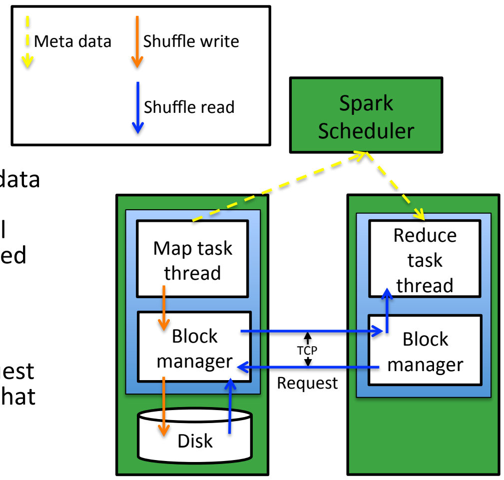
Stages and pipelining

- 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 *stage*
  - **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
  - **Stage**: A pipelined set of operations
    - **Task**: Execution of stage on a single partition
- Every stage ends with a shuffle, an output or returning data back to the driver.

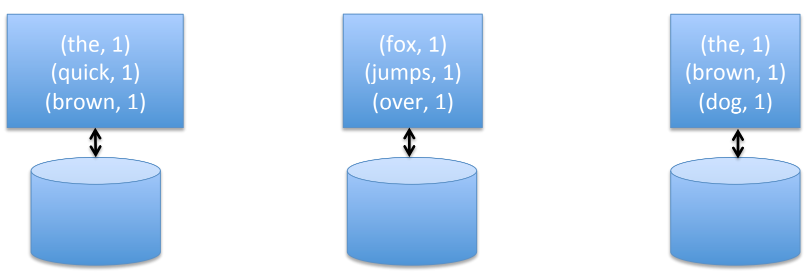
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 67

Shuffle implementation

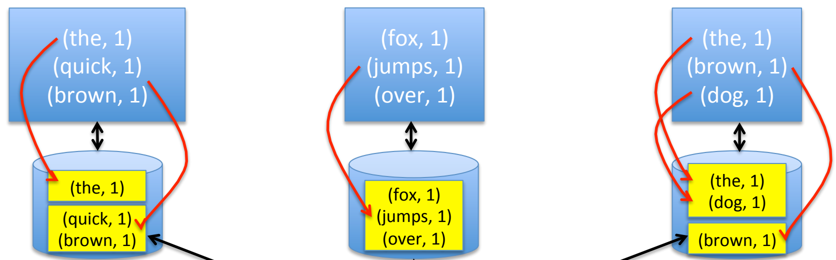
- All data exchanges between executors implemented via shuffle
  - Senders ("mappers") send data to block managers; block managers write to disks, tell scheduler how much destined for each reducer
  - Barrier until all mappers complete shuffle writes
  - Receivers ("reducers") request data from block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



Shuffle Walkthrough

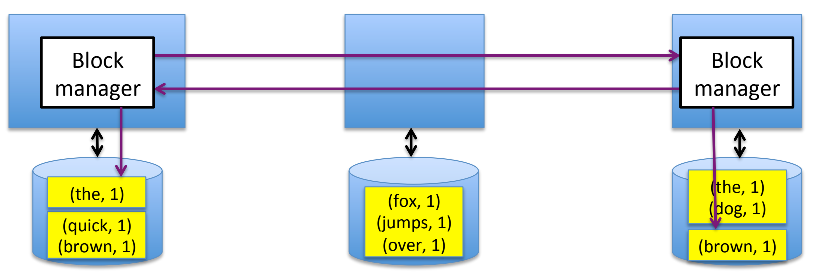


Shuffle Walkthrough



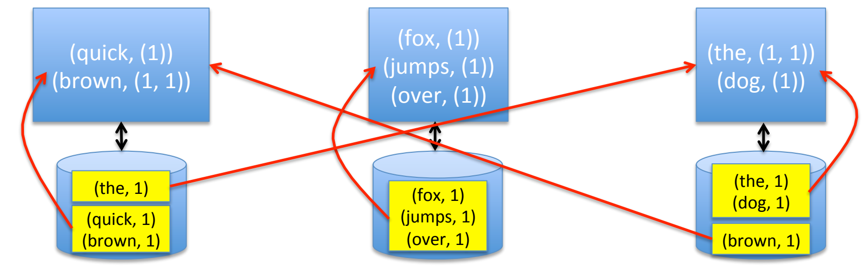
Sort-based shuffle groups by destination "reducer" (receiver).

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
  – Reduces amount of data sent over the network

Discussion

- Common (mis??)conceptions about data analytics performance:
  - Optimize Network
  - Optimize IO
  - Stragglers are tricky
- Many have interpreted the paper you read as claiming these are not accurate
  - Computation is now the real bottleneck
- Do you agree? Why or why not?
  - Do you think that is what the authors were really trying to say?
- Were the workloads representative enough? Does this matter?
- What should we focus on now, to improve performance and scaling?