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 \textit{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
  – \textbf{Volume}: very large amount of data
  – \textbf{Velocity}: data coming in very rapidly
  – \textbf{Variety}: many different types of data
• Two additional V’s are sometimes added:
  – \textbf{Veracity}: Large variance in the quality of the data/difficulty in determining quality
  – \textbf{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

<table>
<thead>
<tr>
<th>Big Data Analytics Challenge</th>
<th>Parallel Computing Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>High volume of data</td>
<td>Many cluster nodes = large amount of memory to store data (DRAM and disks)</td>
</tr>
<tr>
<td>High velocity of data – need to keep up with rapid streams of new data</td>
<td>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.</td>
</tr>
<tr>
<td>Extracting knowledge from large quantities of data</td>
<td>Large datasets tend to scale up well to large numbers of processes. Parallel versions of many common machine learning/statistical algorithms are well studied</td>
</tr>
<tr>
<td>Veracity and Variability</td>
<td>Data cleaning and preparation tasks often parallelize well.</td>
</tr>
</tbody>
</table>
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

**THE CAST**

- People sit in front of me and ask me to read/write data
- There is only ONE of me and I coordinate everything around here
- We store data... there are billions of us spread out even thousands

**WRITING DATA ON HDFS CLUSTER**

- **REQUEST FROM USER**: Let's start with writing some data
- **BLOCK AND REPLICA**: Are you not forgetting something?
- **DIVIDE FILE INTO BLOCKS**: As per, please a) divide the data in 128MB blocks b) copy each block to three places
- **QUESTION**: There is only ONE of me and I coordinate everything around here
- **A client always knows these two things**: BLOCKSIZE: large file is divided into blocks (usually 64 or 128MB)
  - REPLICA FACTOR: each block is stored in multiple locations (usually 3)

**ASK NAME NODE**

- Let's work on the first block
- My NameNode please help me write a 128MB block with replication of 3

**NAME NODE ASSIGNING DATANODES**

- Replication 3: Here, need to find 3 datanodes for the client
  - Question: How do I do that? Will tell you some other time
Hadoop HDFS

Reading Data in HDFS Cluster

Request from User

Writing file in HDFS -- check. What about reading them? Let's ask the client again.

Mr. Client, please read this file for me.

Roger.

Contact NameNode First...

Please give me info on this file.

Filename

I reply (a) list of all blocks for this file, (b) list of datanodes for each block (sorted by distance from client)

Block 1: at DN x1, y1, z1
Block 2: at DN x2, y2, z2
Block 3: at DN x3, y3, z3
...and so on...

Download Data

Download data from the nearest datanode (the first in list)

Please give me block n

DATA for block n

Umm, Question -- What happens when the datanode is dead, or does not have the data, or the data is corrupted...

Actually, HDFS can very elegantly handle these faults and more...
Hadoop HDFS

FAULT TOLERANCE IN HDFS. PART I: TYPES OF FAULTS AND THEIR DETECTION

FAULT I: NODE FAILURE
There are typically three kinds of faults. The first is NODE FAILURE.

FAULT II: COMMUNICATION FAILURE
Second is COMMUNICATION FAILURE (cannot send and receive data).

FAULT III: DATA CORRUPTION
Third is DATA CORRUPTION. Data can be corrupted while sending over network.

DETECTION #1: NODE FAILURES
NOTE: If NameNode is dead, the entire cluster is dead; NameNode is the SINGLE POINT OF FAILURE.

DETECTING DATANODE FAILURE
We send HEARTBEAT message every 3 seconds. This is our way of saying we are alive.

If I don’t get a message in 10 minutes, the datanode is dead to me. (I may be ALIVE and there was only a network failure, but the NameNode treats both as same)

DETECTION #2: NETWORK FAILURES
Whenever data is sent, an ACK is replied by the reciever:

If the ACK is not received (after several retries), the sender assumes that the host is dead, or the network has failed.

DETECTION #3: CORRUPTED DATA
Checksum is sent along with transmitted data.

Moreover, when I store data in hard disks, I also store the checksum.

DETECTING CORRUPTED HARD DRIVES
Periodically, all datanodes send BLOCKREPORT to the nameNode.

BLOCK REPORT
List of all blocks I have

REGAP: HEARTBEAT MESSAGES AND BLOCK REPORTS
We send heartbeat every 3 seconds to say we are alive.

We send block reports and we skip blocks that are corrupted.

(which is how the nameNode will know which blocks are left)
**Hadoop HDFS**

**FAULT TOLERANCE IN HDFS. PART II: HANDLING READING AND WRITING FAILURES**

### HANDLING WRITE FAILURES

- One thing I should have said earlier: I write the block in smaller data units (usually 64KB) called "packets".

Remember replication pipeline?

- Moreover, each datanode replies back an ACK for each packet to confirm that they got it.

ACK

- So, if I don’t get ACKs from some datanode, I know it is dead. I adjust the pipeline to skip him.

### HANDLING READ FAILURES

- Here’s the adjusted pipeline. Note that the block will be "under replicated", but the namenode will take care of that later on.

Remember, when I asked for location of a block, the namenode gave me locations of all datanodes

- If one datanode is dead, I read from the others in the list.

Got Data? No?

- I continually update these two tables:

<table>
<thead>
<tr>
<th>List of Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block 1 - stored at DN1, DN2, DN3</td>
</tr>
</tbody>
</table>

- If I find a block on a datanode is corrupted, I update first table (by removing bad DN from block’s list)

- And if I find that a datanode has died, I update both tables.

### UNDER REPLICATED BLOCKS

- I scan the first list (list of blocks) periodically, and see if there are blocks that are not replicated properly.

- These are called "under replicated" blocks

### Could you copy the block from that datanode?

- Hey, I need to copy a block from you

- Umm, one more question: All of this works if there is at least one valid copy of the block somewhere, right?

That’s correct. HDFS cannot guarantee that at least one replica will always survive. But it tries its best by smartly selecting replica locations, as we will see next —
Hadoop HDFS

**REPLICA PLACEMENT STRATEGY**

**RACKS AND DATANODES**
The cluster is divided into RACKS. Each rack has multiple datanodes.

**SELECTION FIRST REPLICA LOCATION**
First replica location is simple:
- If the writer is a member of cluster, it is selected as first replica
- Otherwise some random datanode is selected

**NEXT TWO REPLICA LOCATIONS**
- Pick a different rack than first replica's
- Select two different datanodes on that rack

**SUBSEQUENT REPLICA LOCATIONS**
- Pick any random datanode, if it satisfies these two conditions:
  - Only one replica per datanode
  - Max two replicas per rack

**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

```java
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);
        }
    }

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

        public void reduce(Text key, Iterable<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            IntWritable resultSum = new IntWritable(0);
            for (IntWritable v : values) {
                resultSum.increment();
            }
            output.collect(key, resultSum);
        }

    }

}
```
MapReduce Word Count

```java
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

```scala
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)
```

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

Create array of (1, 2, ..., 1,000,000)

val arr1M = Array.range(1,1000001)

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

val rdd1M = sc.parallelize(arr1M, 8)

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

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

evens.take(5)

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

Spark action (return result to driver)
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)

Spark action (return result to driver)

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:
{250002, 250004, ...}  
{750002, 750004, ...}

Executor 3:
{375002, 375004, ...}  
{875002, 875004, ...}

val evens = rdd1M.filter(a => a%2==0)
Example: Line-by-line

**Conceptually ...**

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

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, ...}

\[\text{evens.take(5)}\]

Now let's try it out ...

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

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

FilteredRDD 0

...

RDD Partition 7

FilteredRDD 7

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

DAG (Directed Acyclic Graph) schedule

Take Result:
RETURNS DATA

Driver:
{1, ..., 1,000,000}

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

Input: Arr1M

RDD Partition 0

FilteredRDD 0

...

RDD Partition 7

FilteredRDD 7

evens.take(5)

DAG (Directed Acyclic Graph) schedule
What's going on here?

Lazy Evaluation:

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

evens.take(5)

DAG (Directed Acyclic Graph) schedule

Start computing!

Spring 2017
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

FilteredRDD 0

RDD Partition 7

FilteredRDD 7

Take Result: RETURNS DATA

evens.take(5)

DAG (Directed Acyclic Graph) schedule

Modified example

```scala
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

- 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...

```scala
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)
```

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

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

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

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.

Modified example

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

```scala
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

```scala
case class Pair(v: String, n: Int)
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

```scala
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

```scala
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

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

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

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

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
"the quick brown fox jumps over the brown dog"

HDFS Blocks
- Block 1
- Block 2
- Block N

Partitions
- Partition 1
- Partition 2
- Partition N

Splits
- Split 1
- Split 2
- Split N

Pairs
- Pair 1
- Pair 2
- Pair N

Groups
- Group 1
- Group 2
- Group N

Counts
- Count 1
- Count 2
- Count N

No cross-node dependencies: operations pipelined into single task.
Execution

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 → Group N → Count N

Spring 2017
UW CSEP 590 (PMP Programming Systems):
Ringenburg
These are also pipelined into a single task per node.
Stages and pipelining

• If an RDD partitions's dependencies are on a single other RDD partition (or on co-partitioned 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.
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	implementaHon
\[\begin{array}{c}
\text{Shuffle write} \\
\text{Shuffle read}
\end{array}\]
```
Shuffle Walkthrough

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

"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

```scala
val lines = sc.textFile("mytext")
val words = lines.flatMap (line => line.split(" "))
val wordKV = words.map(s => (s, 1))
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?