



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?

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

2



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

3



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

4



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

5



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

6



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.

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

7



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

8



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.

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

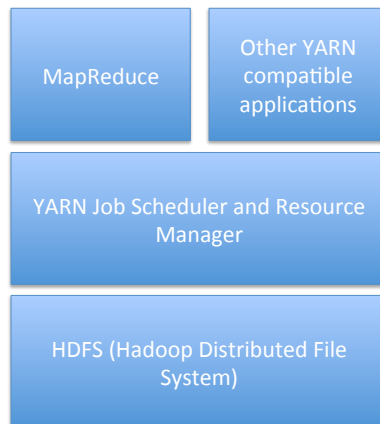
9



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 DISTRIBUTED FILE SYSTEM (HDFS)

THE CAST

CLIENT: People sit in front of me and ask me to read/write data

NAMENODE: There is only ONE of me... ..and I coordinate everything around here

DATANODES: We store data... ..there are MANY of us sometimes even thousands!

WRITING DATA IN HDFS CLUSTER

REQUEST FROM USER

Let's start with writing some data..

Mr. Client, please write 200 MB data for me

It'll be my pleasure. But--

BLOCK AND REPLICATION

--are you not forgetting something?

Ah yes.. please:

a) divide the data in 128MB blocks

b) copy each block in three places

A good client always knows these two things:

BLOCKSIZE: large file is divided in blocks (usually 64 or 128MB)

REPLICATION FACTOR: each block is stored in multiple locations (usually 3)

DIVIDE FILE INTO BLOCKS

First-- I divide the big file into blocks

ASK NAMENODE

Lets work on the first block first



Mr. Namenode, please help me write a 128MB block with replication of 3

NAMENODE ASSIGNS DATANODES

Replication 3.. Hmm, need to find 3 datanodes for this client

How do I do that? Will tell you some other time

Hadoop HDFS

© Maneesh Varshney. mvarshney@gmail.com

Here you go buddy. Addresses of three datanodes. I have also sorted them in increasing distance from you

thanks!

Datanode 1, Datanode 2, Datanode 3

CLIENT STARTS WRITING DATA

I send my data (and the list) to first datanode only

I store the data in hard drive, and--

WHILE I am receiving data, I forward the same data to the next datanode

I'll do the same what previous guy did

I am the last in chain..

TA..DA.. REPLICATION PIPELINE

INFORM NAMENODE WHEN DONE

Once all data (for this block) is written to hard disk send DONE to namenode

Done Done Done

Block successfully stored and replicated in HDFS

When I am done with a block, I repeat the same steps with remaining blocks

WHEN ALL BLOCKS ARE WRITTEN..

All blocks written, please close file

Case closed !! NOW I store all meta information in persistent storage (hard disks)

RECAP



I divided the file in blocks--

--for each block, I provided address of datanodes--

--we stored data via Replication Pipeline

12

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...

Now I know how many blocks to download, and the datanodes where each block is stored

So I download each block, in turn, like so --

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 as we will see next --

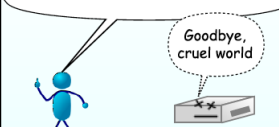
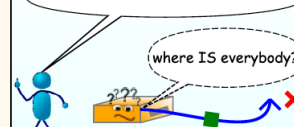
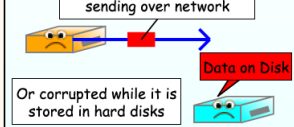
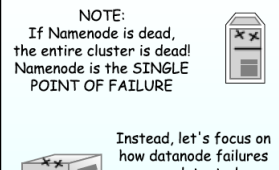

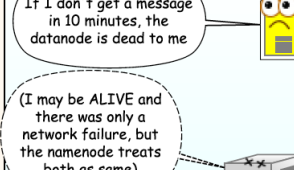
Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
13




Hadoop HDFS




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

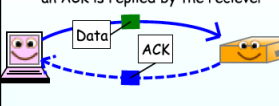
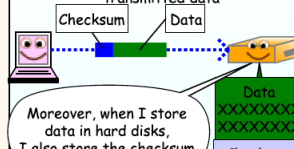
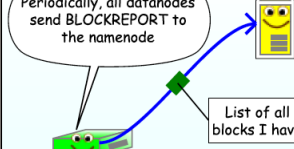
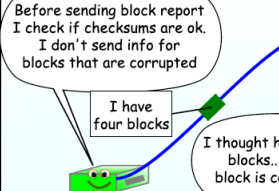
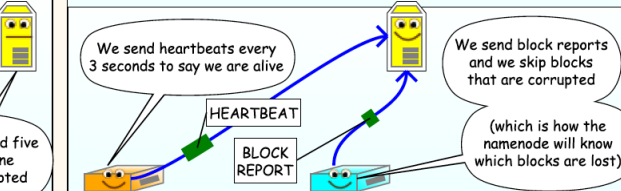
<p>FAULT I: NODE FAILURE</p> <p>There are typically three kinds of faults: The first is NODE FAILURE</p> <p>Goodbye, cruel world</p> 	<p>FAULT II: COMMUNICATION FAILURE</p> <p>Second is COMMUNICATION FAILURE (cannot send and receive data)</p> <p>where IS everybody?</p> 	<p>FAULT III: DATA CORRUPTION</p> <p>Third is DATA CORRUPTION</p> <p>Data can be corrupted while sending over network</p> <p>Or corrupted while it is stored in hard disks</p> 
<p>DETECTION #1: NODE FAILURES</p> <p>NOTE: If Namenode is dead, the entire cluster is dead! Namenode is the SINGLE POINT OF FAILURE</p> <p>Instead, let's focus on how datanode failures are detected</p> 	<p>DETECTING DATANODE FAILURE</p> <p>We send HEARTBEAT message every 3 seconds. This is our way of saying we are alive</p> 	<p>If I don't get a message in 10 minutes, the datanode is dead to me</p> <p>(I may be ALIVE and there was only a network failure, but the namenode treats both as same)</p> 

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
14




Hadoop HDFS




<p>DETECTION #2: NETWORK FAILURES</p> <p>Whenever data is sent, an ACK is replied by the receiver</p> <p>If the ACK is not received (after several retries), the sender assumes that the host is dead, or the network has failed</p> 	<p>DETECTION #3: CORRUPTED DATA</p> <p>Checksum is sent along with transmitted data</p> <p>Moreover, when I store data in hard disks, I also store the checksum</p> 	<p>DETECTING CORRUPTED HARD DRIVES</p> <p>Periodically, all datanodes send BLOCKREPORT to the namenode</p> <p>List of all blocks I have</p> 
<p>Before sending block report I check if checksums are ok. I don't send info for blocks that are corrupted</p> <p>I have four blocks</p> <p>I thought he had five blocks.. so one block is corrupted</p> 	<p>RECAP: HEARTBEAT MESSAGES AND BLOCK REPORTS</p> <p>We send heartbeats every 3 seconds to say we are alive</p> <p>We send block reports and we skip blocks that are corrupted</p> <p>(which is how the namenode will know which blocks are lost)</p> 	

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
15



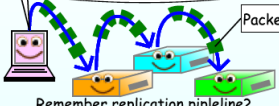
Hadoop HDFS



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"



Packet

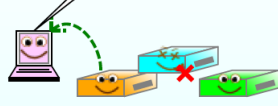
Remember replication pipeline?

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

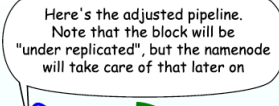


ACK

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

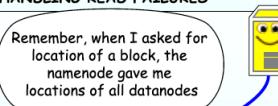


Here's the adjusted pipeline. Note that the block will be "under replicated", but the namenode will take care of that later on



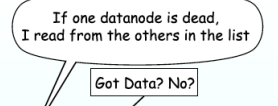
HANDLING READ FAILURES

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




DN 1, DN 2, DN 3

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




Got Data? No?
Got Data?

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
16




Hadoop HDFS



FAULT TOLERANCE IN HDFS. PART III: HANDLING DATANODE FAILURES

First-- I must tell you about the two tables I keep..



List of Blocks

Block 1 - stored at DN1, DN2, DN3


Block 2 - stored at DN1, DN4, DN5

List of Datanodes

Datanode 1 - has block 1, 2, ...

Datanode 2 - has block 1, 5, ...


I continuously update these two tables--



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

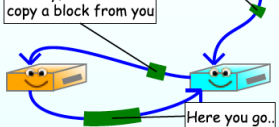
For all under-replicated blocks, I ask other datanodes to copy them from datanodes that have the replica like so --



Could you copy the block from that datanode




Hey, I need to copy a block from you




Here you go..

Umm.. one more question: All of this works if there is atleast one valid copy of the block somewhere.. right?




That's correct. HDFS cannot guarantee that atleast one replica will always survive. But it tries it best by smartly selecting replica locations, as we will see next --

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
17



Hadoop HDFS




REPLICA PLACEMENT STRATEGY

Remember I promised to tell you how I select datanode locations for storing the replicas of a block?


Hang tight.. here it goes..

RACKS AND DATANODES


The cluster is divided into RACKS
Each rack has multiple datanodes



Rack 1



Rack 2



Rack 3

SELECTING 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 datanode on that rack




first replica



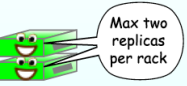
next two replicas

SUBSEQUENT REPLICA LOCATIONS

Pick any random datanode, if it satisfies these two conditions:



Only one replica per datanode




Max two replicas per rack


Please note the fine print: sometimes those two conditions cannot be satisfied, in which case they are .. ahem.. ignored (convenient eh?)

Also, HDFS allows you use your own placement algorithm. So if you know a better algorithm, don't be shy now...

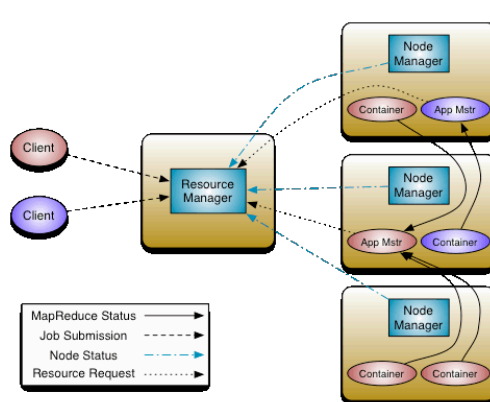
- Cartoon ©Maneesh Varshney (<https://drive.google.com/file/d/0B-zw6KHOtbT4MmRkZWJyZEtYjI3Ni00NTFjLWWE0OGItYTU5OGMxYjc0N2M1/view>)



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



MapReduce Status →

Job Submission - - - - -

Node Status - - - - -

Resource Request ·····

From hadoop.apache.org

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

19



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

20



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);
        }
    }
}
```

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

21



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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

24



Hadoop Performance Issues



- Conventional wisdom: gated on IO bandwidth, network interconnect bandwidth
 - Shuffle: Writing mapper output to disk, sending over network, reading reducer input from 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

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

25



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)

Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

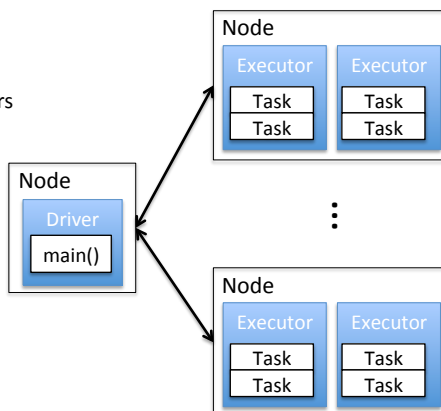
26



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



Spring 2017

UW CSEP 590 (PMP Programming Systems):
Ringenburg

27



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)

```

Spring 2017 UW CSEP 590 (PMP Programming Systems):
Ringenburg 30



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)

```

Spring 2017 UW CSEP 590 (PMP Programming Systems):
Ringenburg 31



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): Ringenb 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): Ringenb 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


Spring 2017

UW CSEP 590 (PMP Programming Systems): Ringenbun

34



Example: Line-by-line



Conceptually ...

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

Executor 0:

Executor 1:

Executor 2:

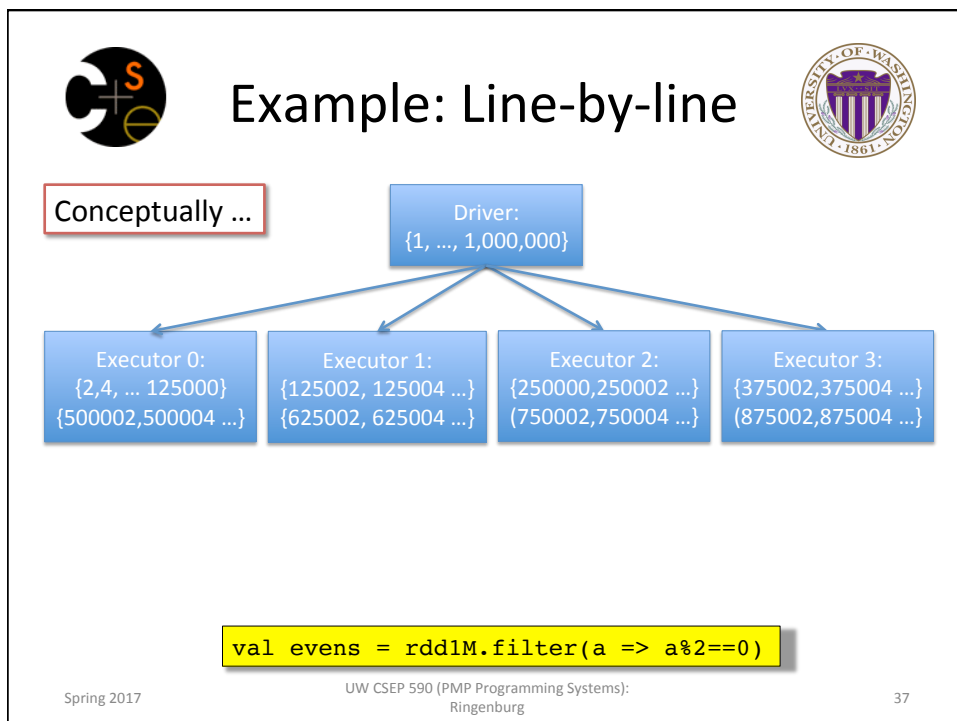
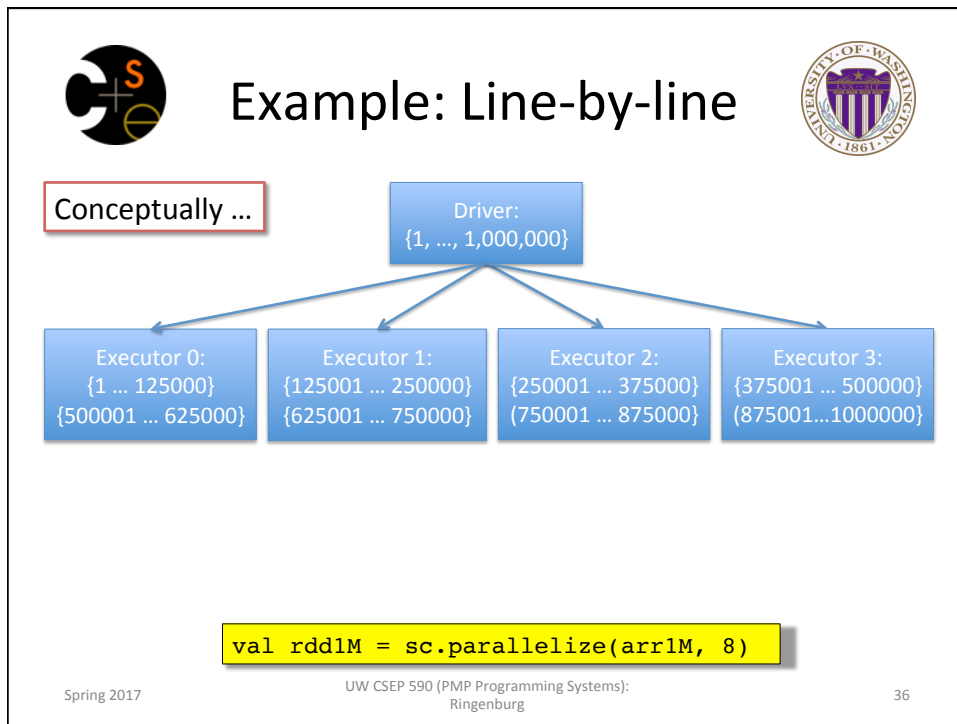
Executor 3:

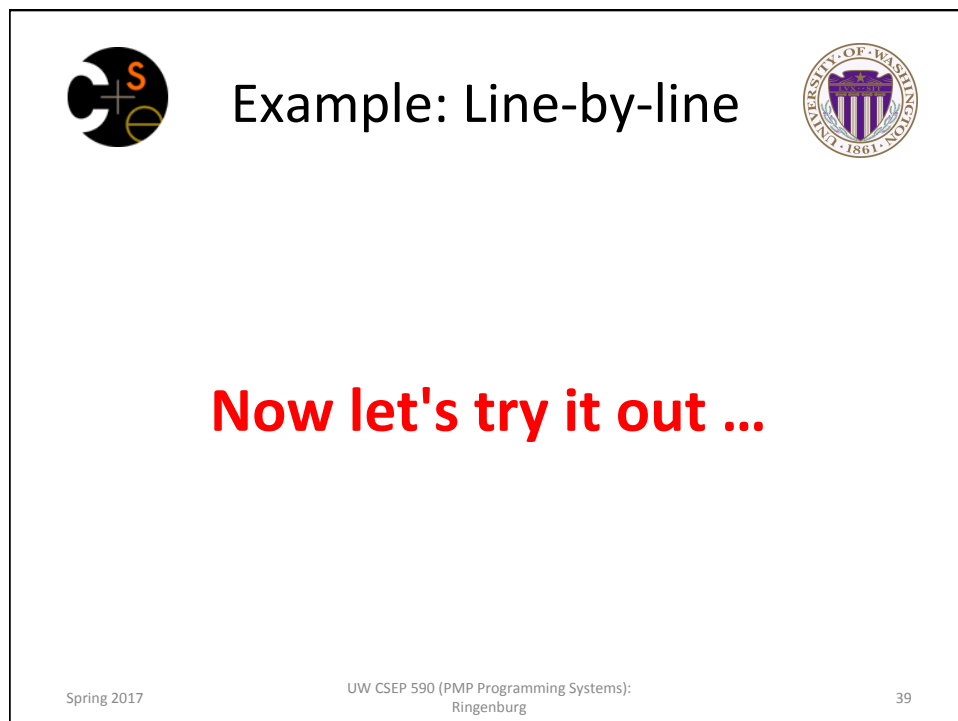
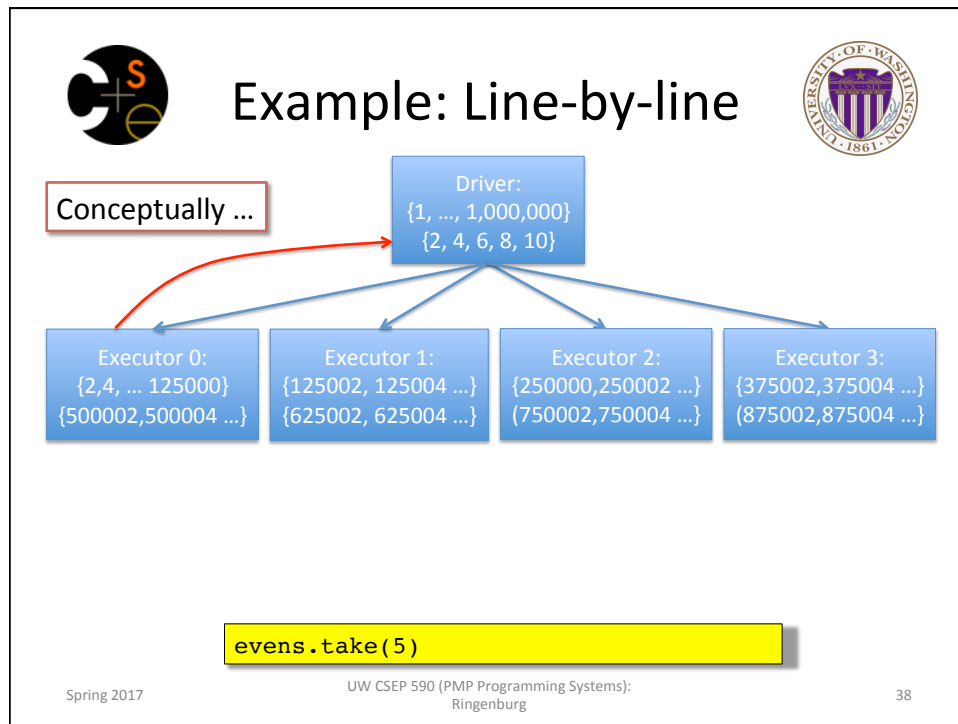
val arr1M = Array.range(1,1000001)



Spring 2017

UW CSEP 590 (PMP Programming Systems): Ringenbun

35





What's going on here?



Lazy Evaluation:

```

    graph TD
      Driver["Driver:  
{1, ..., 1,000,000}"]
      E0["Executor 0:"]
      E1["Executor 1:"]
      E2["Executor 2:"]
      E3["Executor 3:"]
      Driver --> E0
      Driver --> E1
      Driver --> E2
      Driver --> E3
    
```

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

Spring 2017 UW CSEP 590 (PMP Programming Systems):
Ringenburg 40

What's going on here?

Lazy Evaluation:



```

    graph TD
      Driver["Driver:  
{1, ..., 1,000,000}"]
      E0["Executor 0:"]
      E1["Executor 1:"]
      E2["Executor 2:"]
      E3["Executor 3:"]
      Driver --> E0
      Driver --> E1
      Driver --> E2
      Driver --> E3
      Input["Input: Arr1M"] --> RP0["RDD Partition 0"]
      Input --> RP7["RDD Partition 7"]
      RP0 -.- RP7
    
```

DAG (Directed Acyclic Graph) schedule

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

Spring 2017 UW CSEP 590 (PMP Programming Systems):
Ringenburg 41

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

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

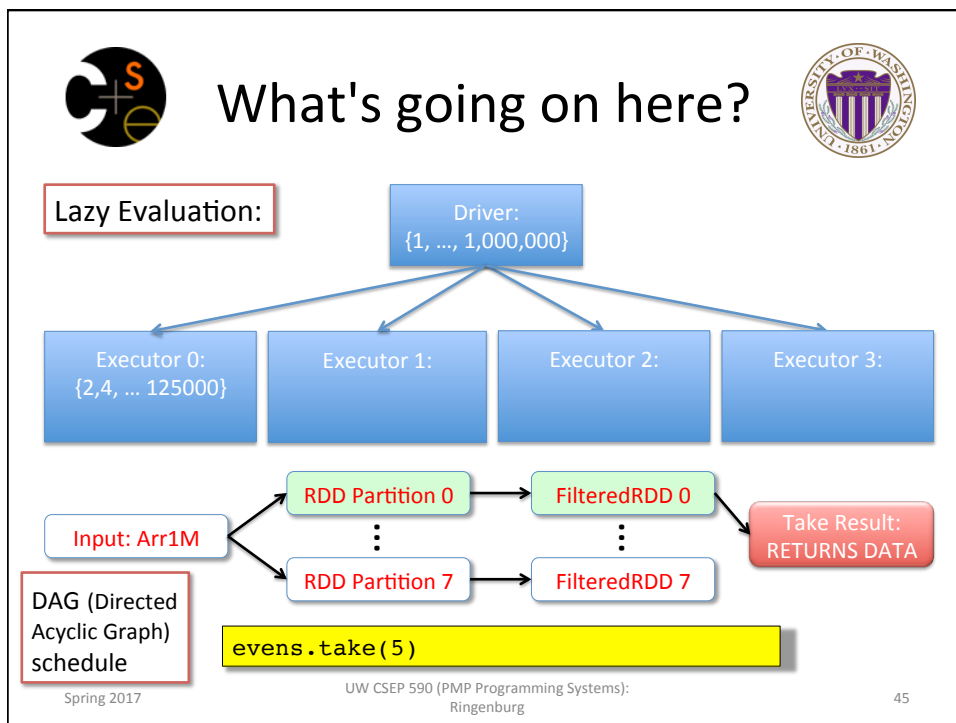
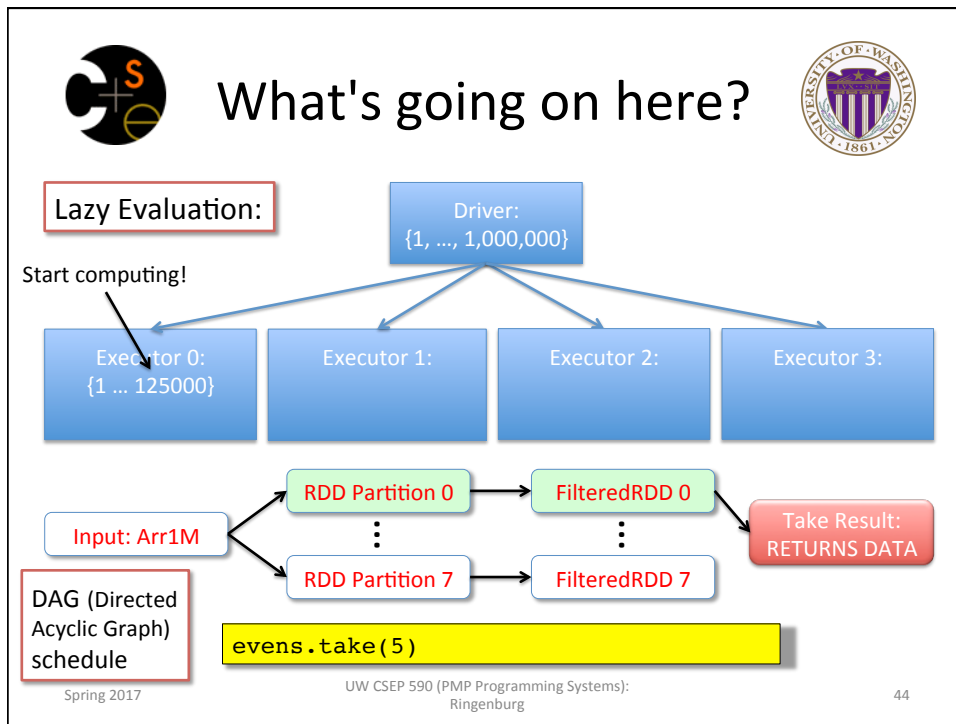
RDD Partition 0 → FilteredRDD 0
 ⋮
 RDD Partition 7 → FilteredRDD 7

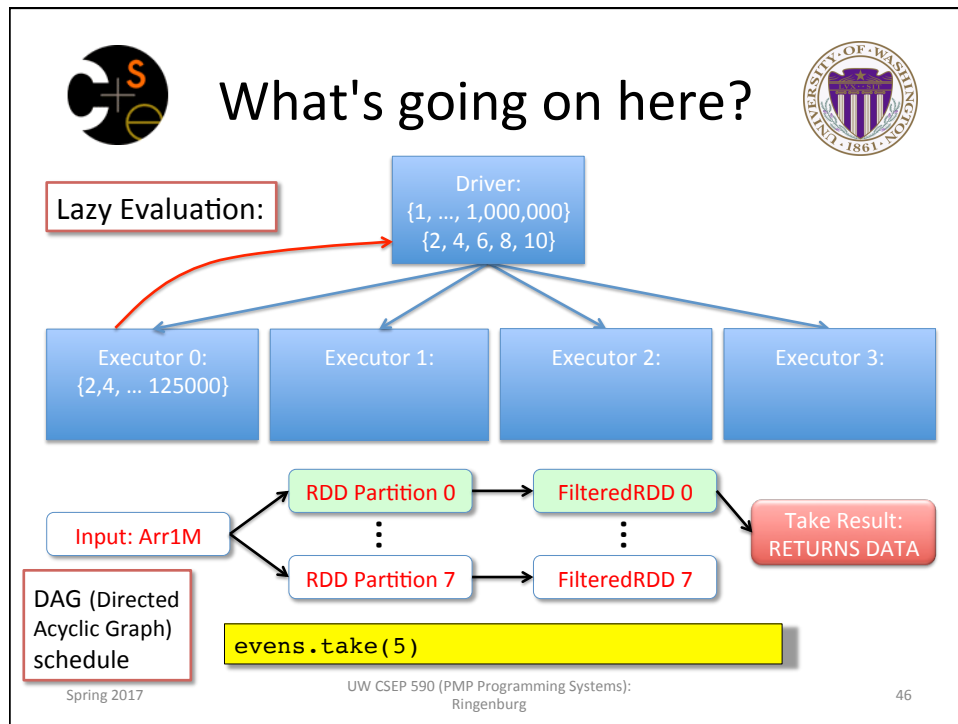
DAG (Directed Acyclic Graph) schedule

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

Take Result: RETURNS DATA

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 43





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...

Spring 2017

UW CSEP 590 (PMP Programming Systems):
 Ringenbun

47

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...

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 48

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...

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 49

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.

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 50

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!

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 51



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 partitions 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

Spring 2017
UW CSEP 590 (PMP Programming Systems):
Ringenburg
54



Communication Example



Load file, default number of partitions 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

Spring 2017
UW CSEP 590 (PMP Programming Systems):
Ringenburg
55



Communication Example



Load file, default number of partitions 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

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
56



Communication Example



Load file, default number of partitions 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

Spring 2017
UW CSEP 590 (PMP Programming Systems): Ringenburg
57



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

HDFS Block 1

⋮

HDFS Block N

→ Partition 1

⋮

→ Partition N

→ Split 1

⋮

→ Split N

→ Pair 1

⋮

→ Pair N

→ Group 1

⋮

→ Group N

→ Count 1


⋮

→ Count N


Spring 2017


UW CSEP 590 (PMP Programming Systems):
Ringenburg

58




Execution

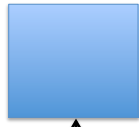




"the quick brown"



"fox jumps over"



"the brown dog"

HDFS Block 1

⋮

HDFS Block N

→ Partition 1

⋮

→ Partition N

→ Split 1

⋮

→ Split N

→ Pair 1

⋮

→ Pair N

→ Group 1

⋮

→ Group N

→ Count 1

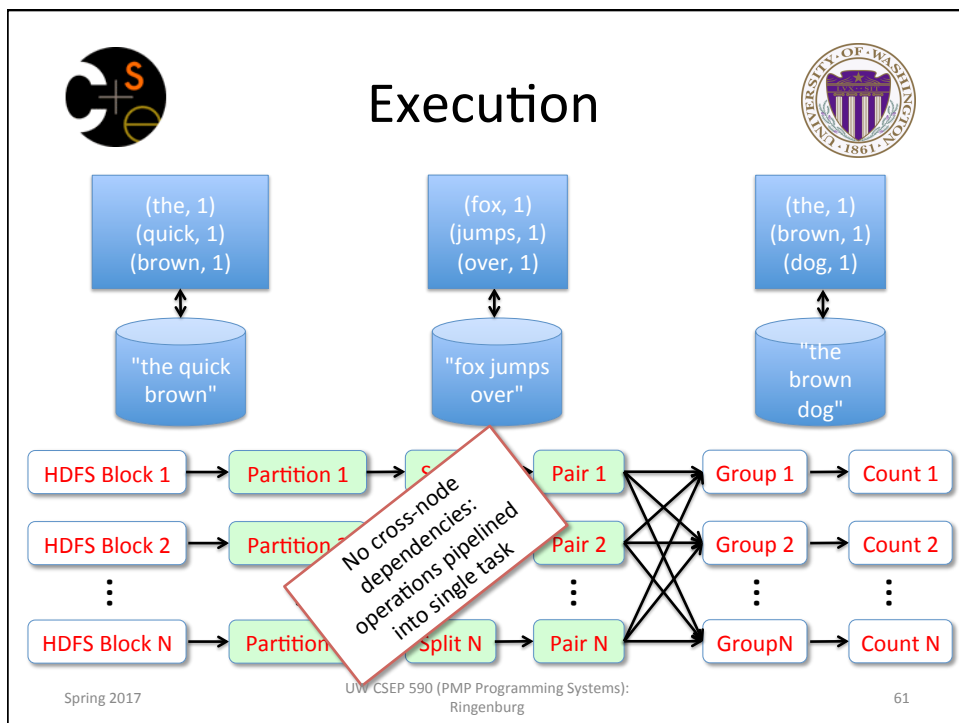
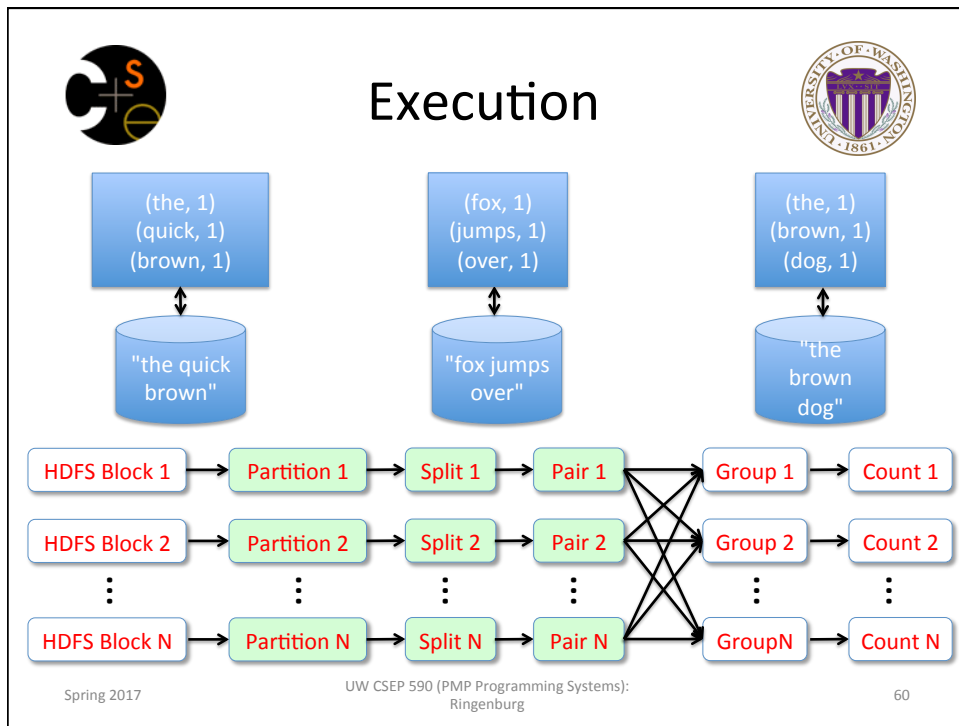
⋮

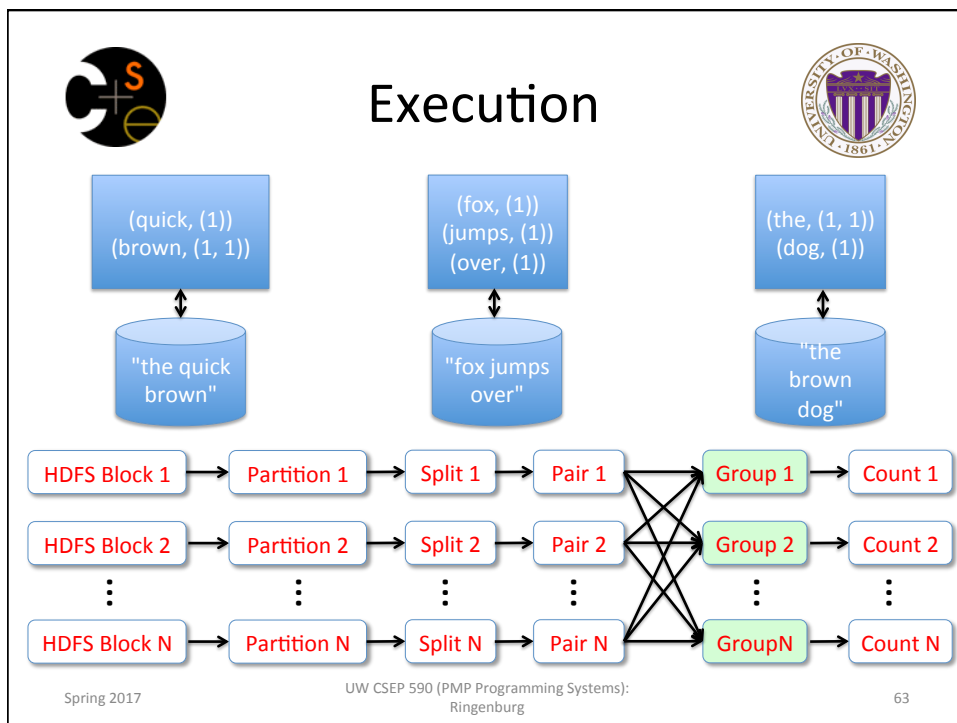
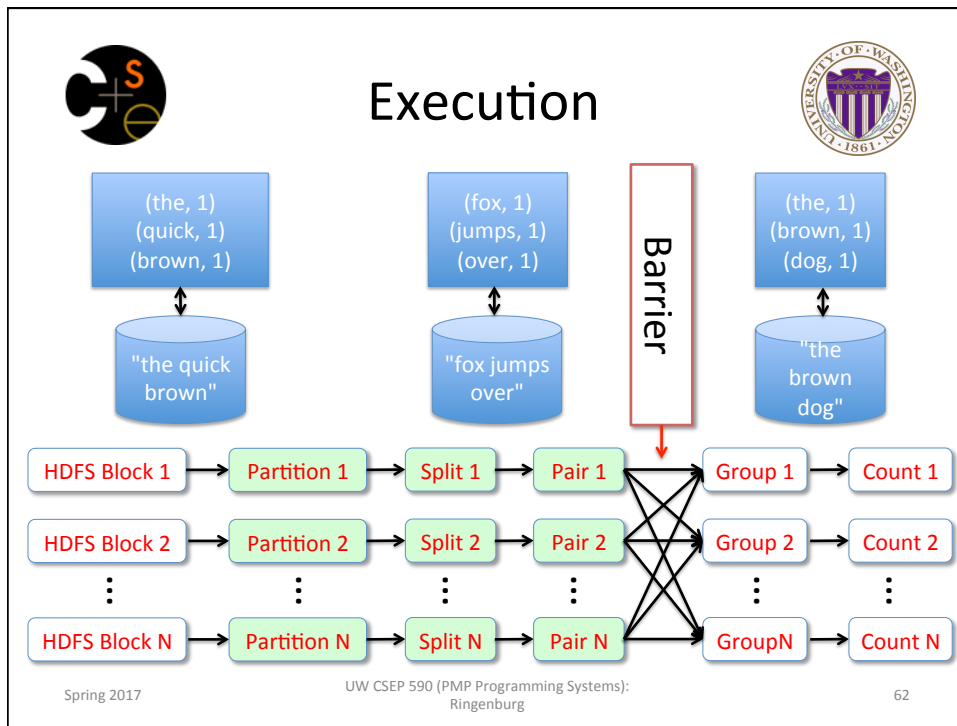
→ Count N

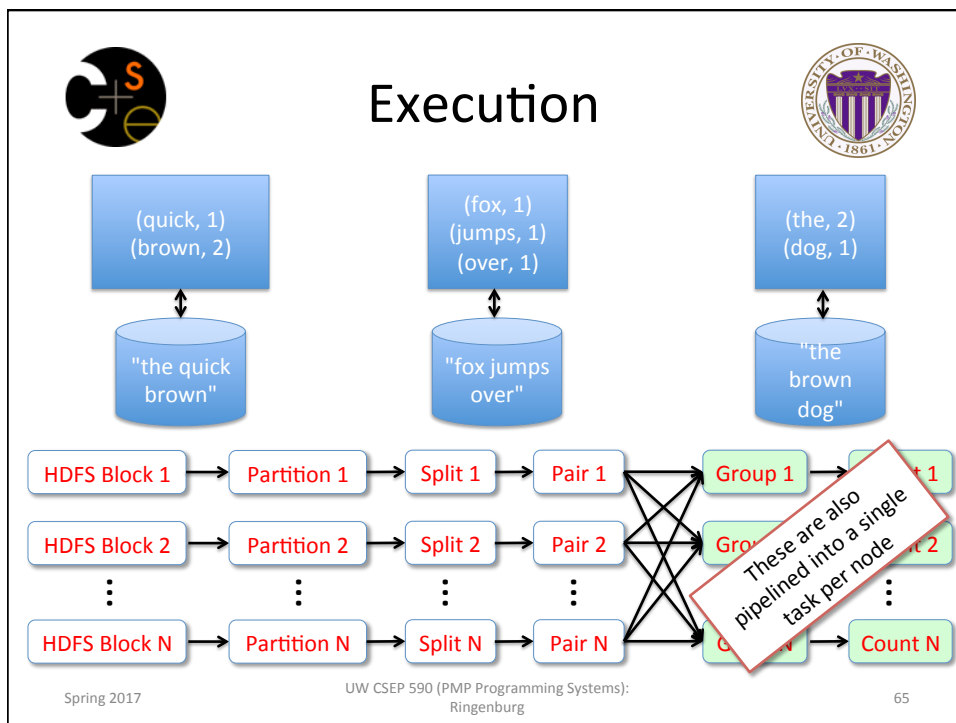
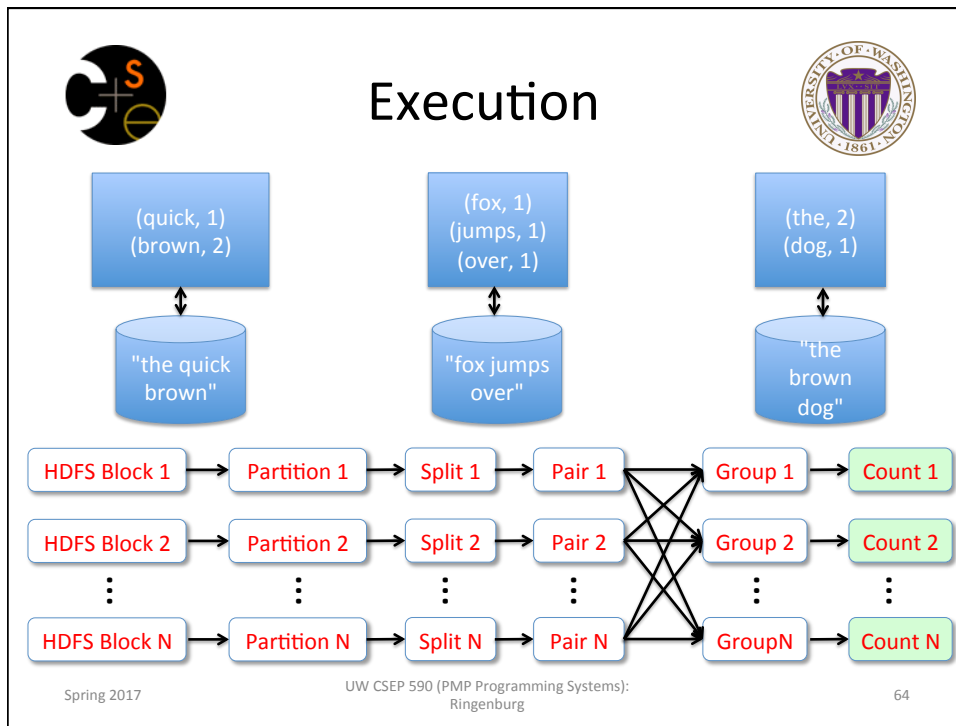
Spring 2017

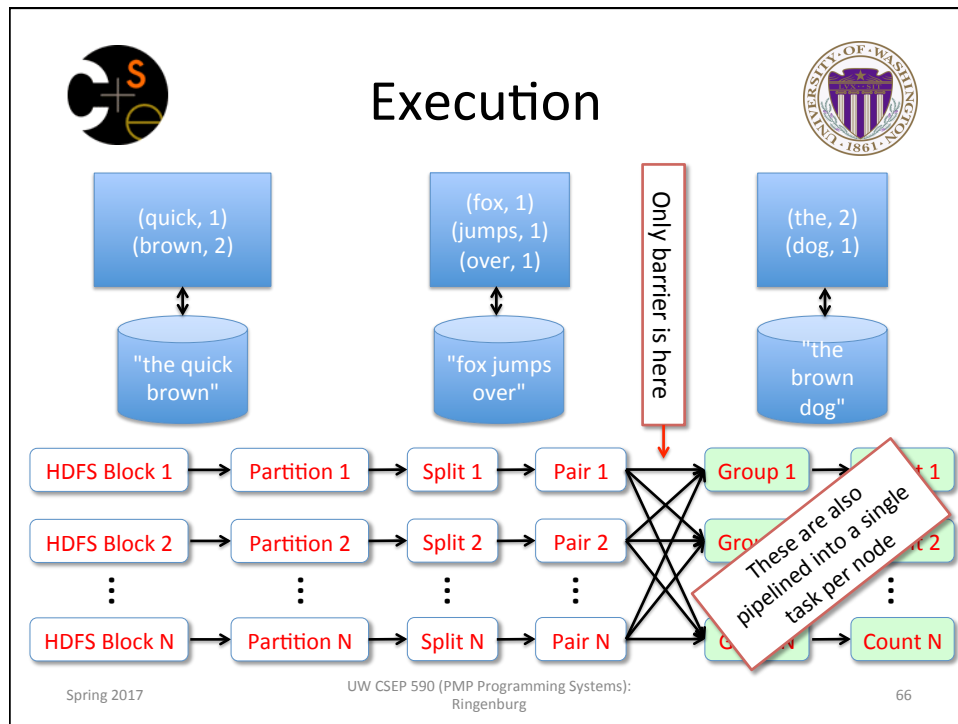
UW CSEP 590 (PMP Programming Systems):
Ringenburg

59









Stages and pipelining

- If an RDD partition'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.

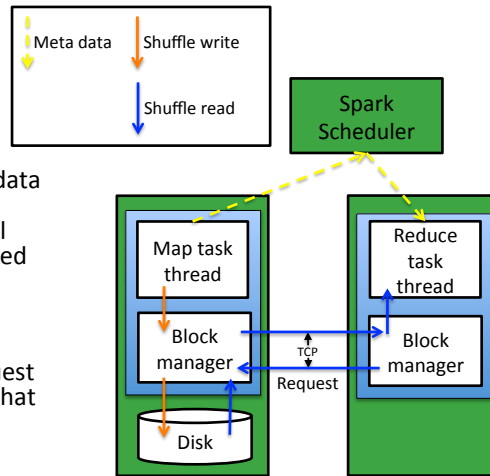
Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenbun 67



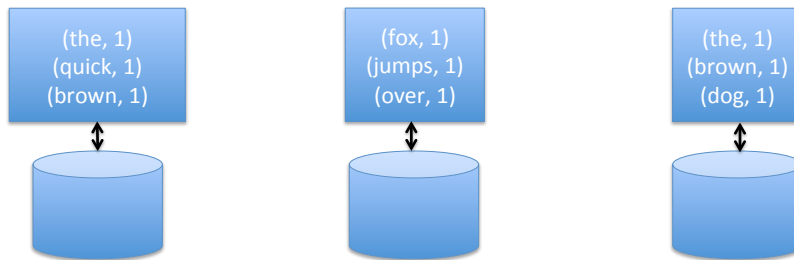
Shuffle implementation

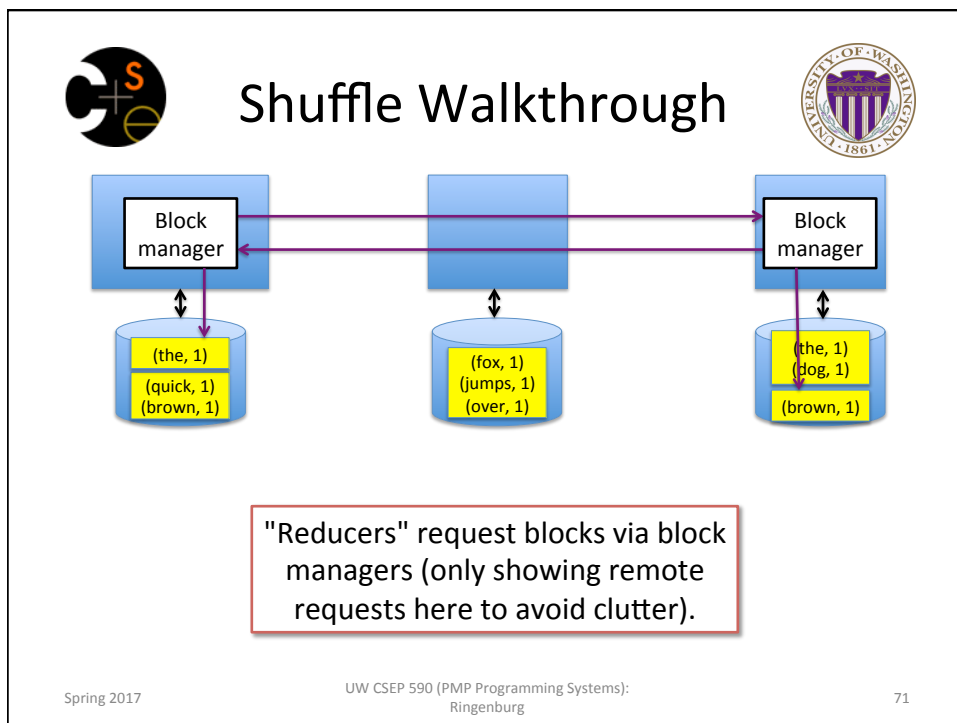
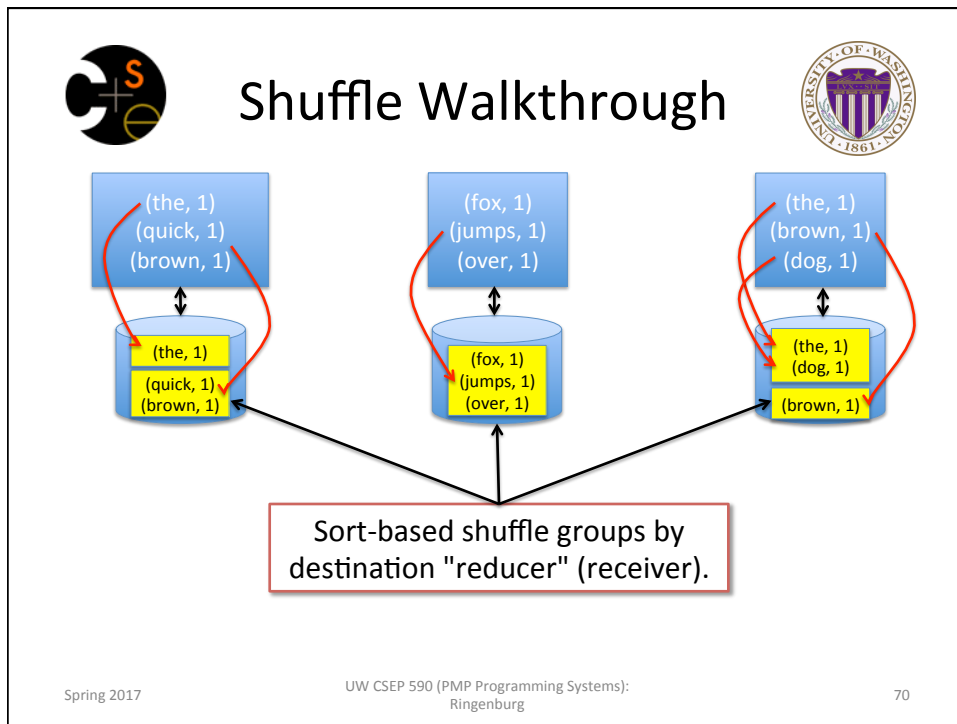


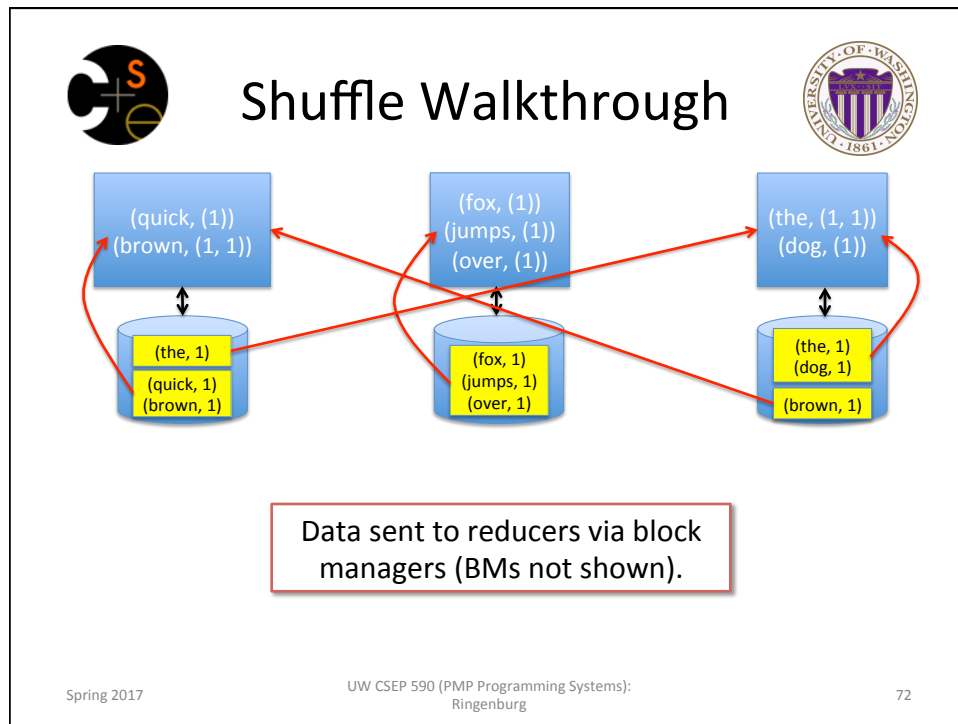
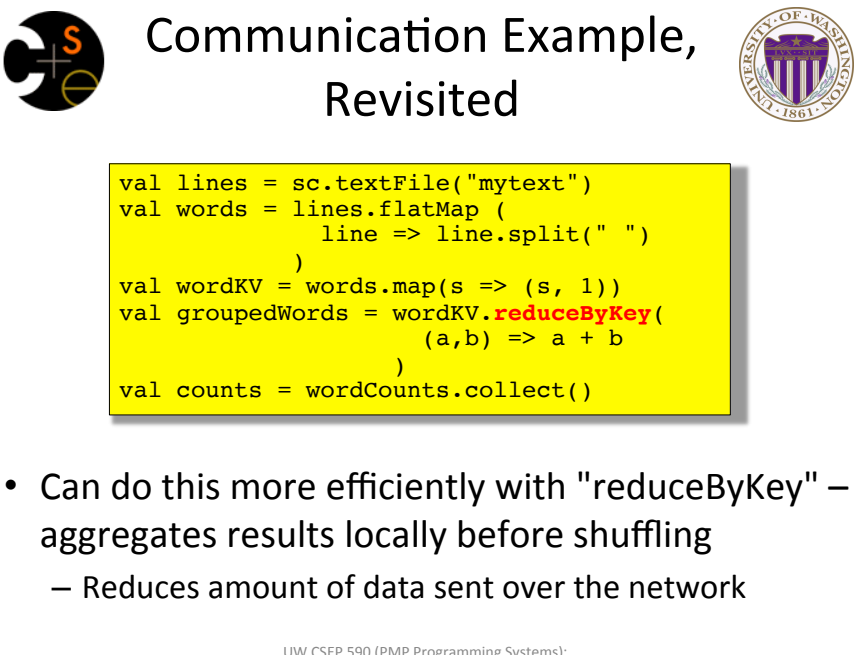
- All data exchanges between executors implemented via shuffle
 - Senders (“mappers”) send data to block managers; block managers write to disks, tell scheduler how much destined for each reducer
 - Barrier until all mappers complete shuffle writes
 - Receivers (“reducers”) request data from block managers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



Shuffle Walkthrough





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

Spring 2017 UW CSEP 590 (PMP Programming Systems): Ringenburg 73



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?