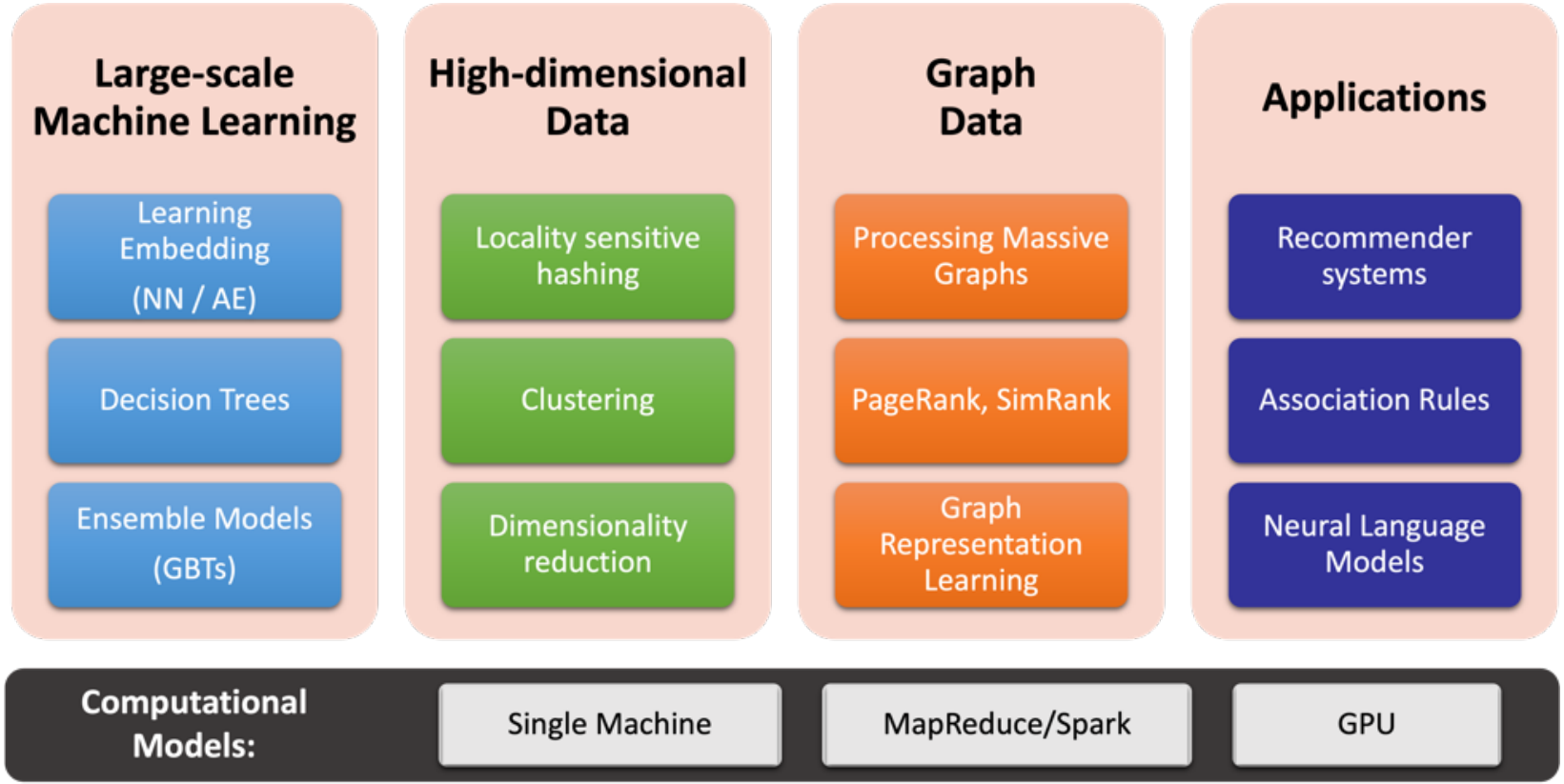
MIE524 Data Mining

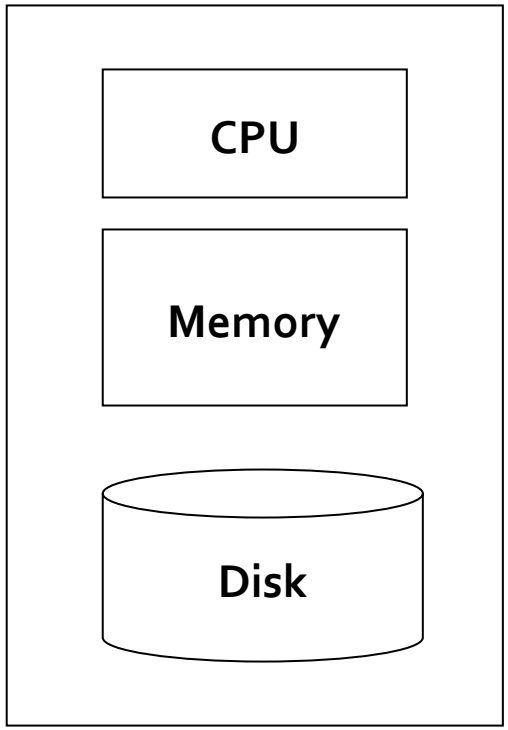
MapReduce and Spark

Slides Credits:
Slides from Leskovec, Rajaraman, Ullman (http://www.mmds.org),
Fatahalian & Olukotun (Stanford CS149) and Kishore Pusukuri (Cornell CS5412)

MIE524: Course Topics (Tentative)



Single Node Architecture



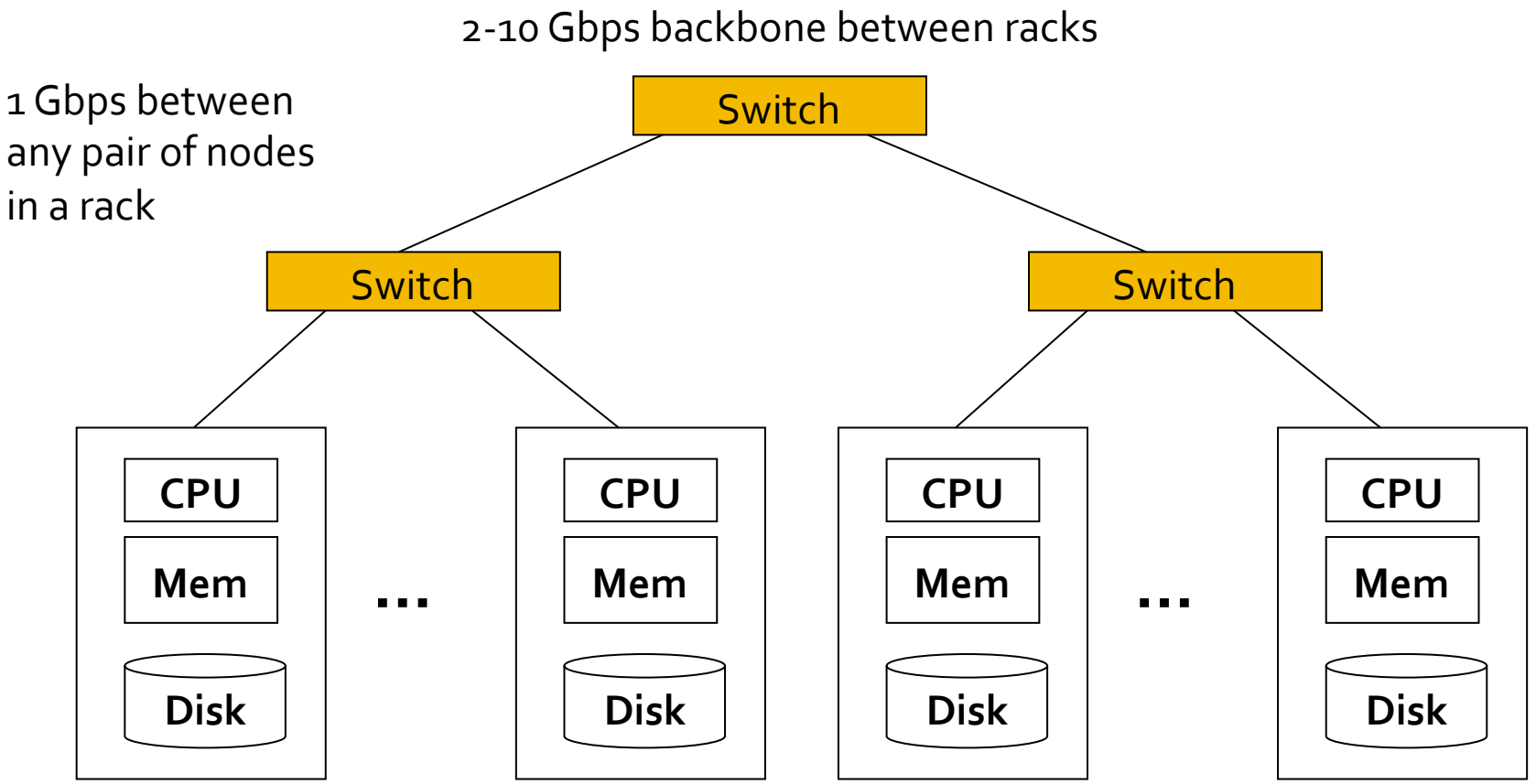
Machine Learning, Statistics

"Classical" Data Mining

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

Cluster Architecture



Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Large-scale Computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**
- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to lose 1/day
    - With 1M machines 1,000 machines fail every day!

An Idea and a Solution

- **Issue:**
  Copying data over a network takes time
- **Idea:**
  - Bring computation to data
  - Store files multiple times for reliability
- **Spark/Hadoop** address these problems
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - MapReduce
    - Spark

Storage Infrastructure

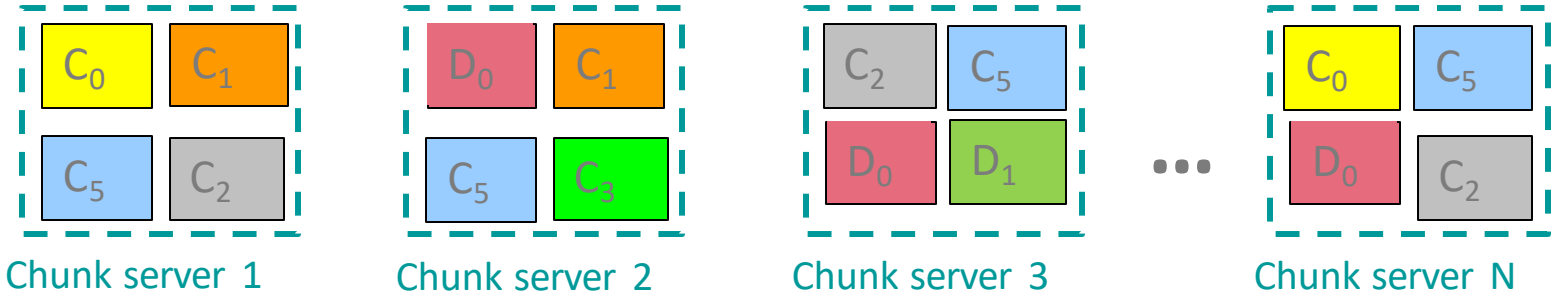
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System**
    - Provides global file namespace
- **Typical usage pattern:**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Master nodes are typically more robust to hardware failure and run critical cluster services.
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed File System

- **Reliable distributed file system**
- Data kept in "chunks" spread across machines
- Each chunk **replicated** on different machines
  - Seamless recovery from disk or machine failure



Notation: C_2… chunk no. 2 of file C

Bring computation directly to the data!

Chunk servers also serve as compute servers

MapReduce: Early Distributed Computing Programming Model

Programming Model: MapReduce

- **MapReduce is a style of programming designed for:**
  1. Easy parallel programming
  2. Invisible management of hardware and software failures
  3. Easy management of very-large-scale data

- It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called "MapReduce"

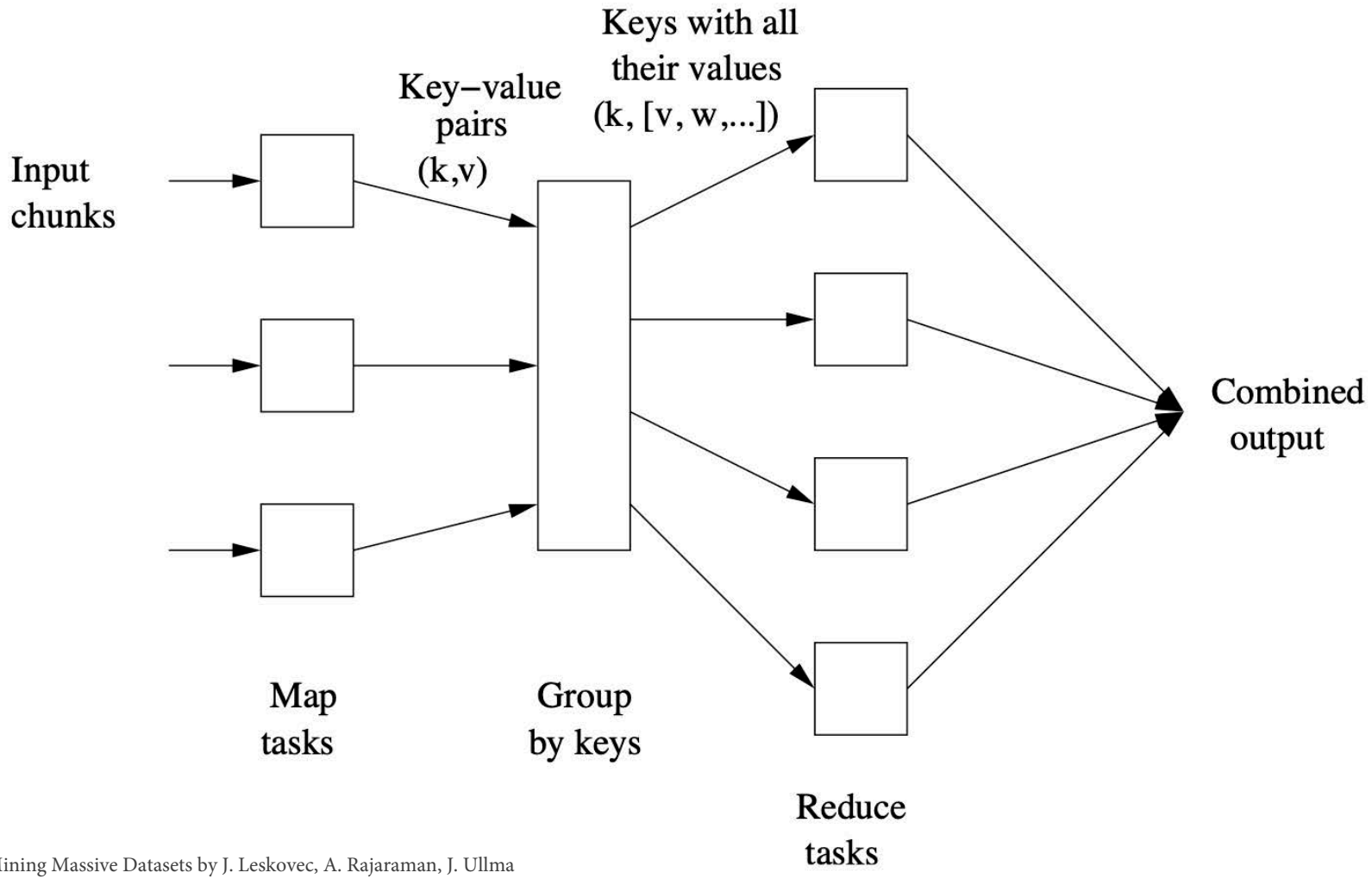
MapReduce: Overview

3 steps of MapReduce

- **Map:**
  - Apply a user-written *Map function* to each input element
    - *Mapper* applies the Map function to a single element
      - Many mappers grouped in a *Map task* (the unit of parallelism)
  - The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- **Group by key:** Sort and shuffle
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs
- **Reduce:**
  - User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, **Map** and **Reduce** change to fit the problem

MapReduce Pattern



Mining Massive Datasets by J. Leskovec, A. Rajaraman, J. Ullma

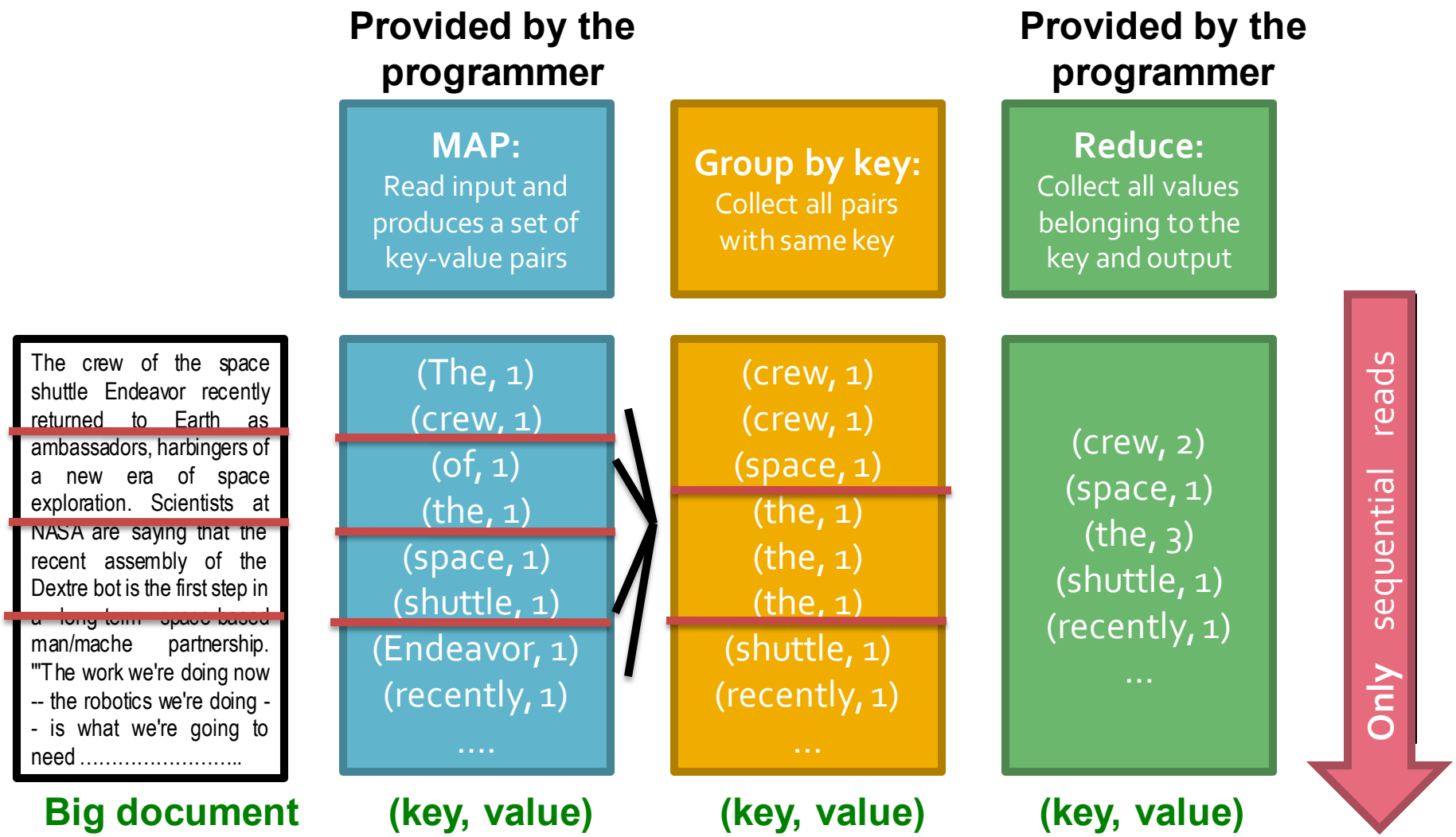
Example: Word Counting

Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file

- **Many applications of this:**
  - Analyze web server logs to find popular URLs
  - Statistical machine translation:
    - Need to count number of times every 5-word sequence occurs in a large corpus of documents

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
# key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

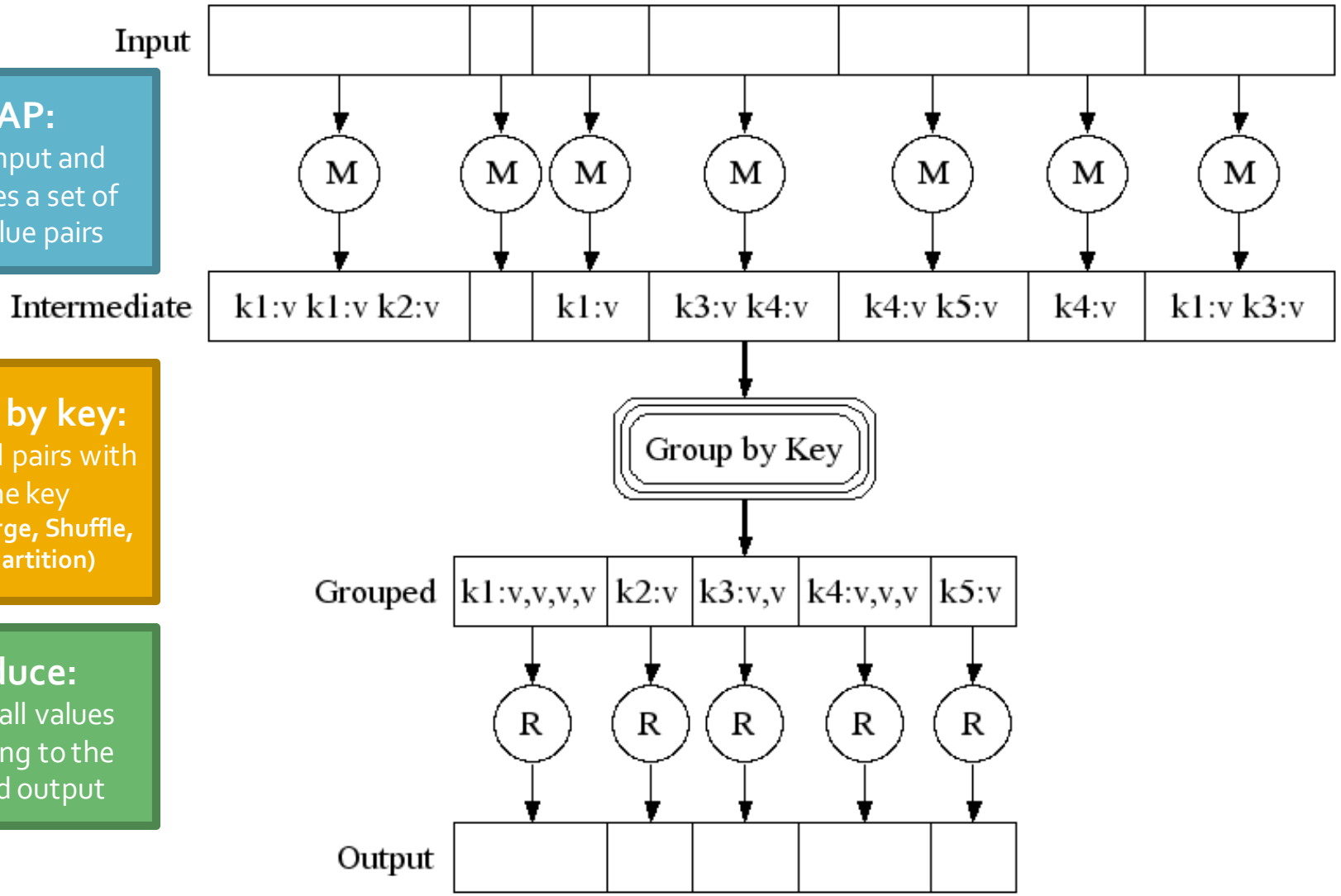
```
reduce(key, values):
# key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

Map-Reduce: A diagram

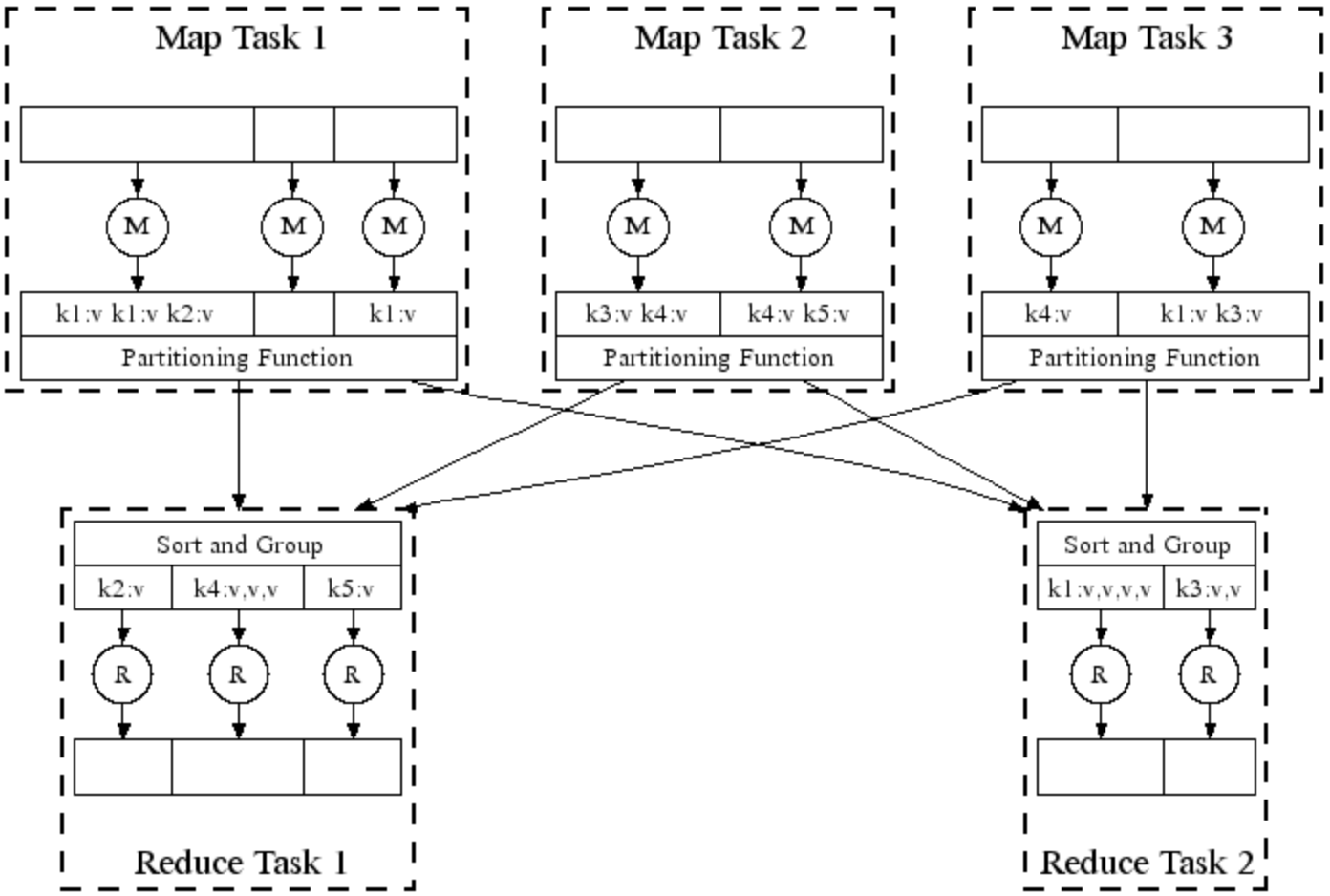
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output



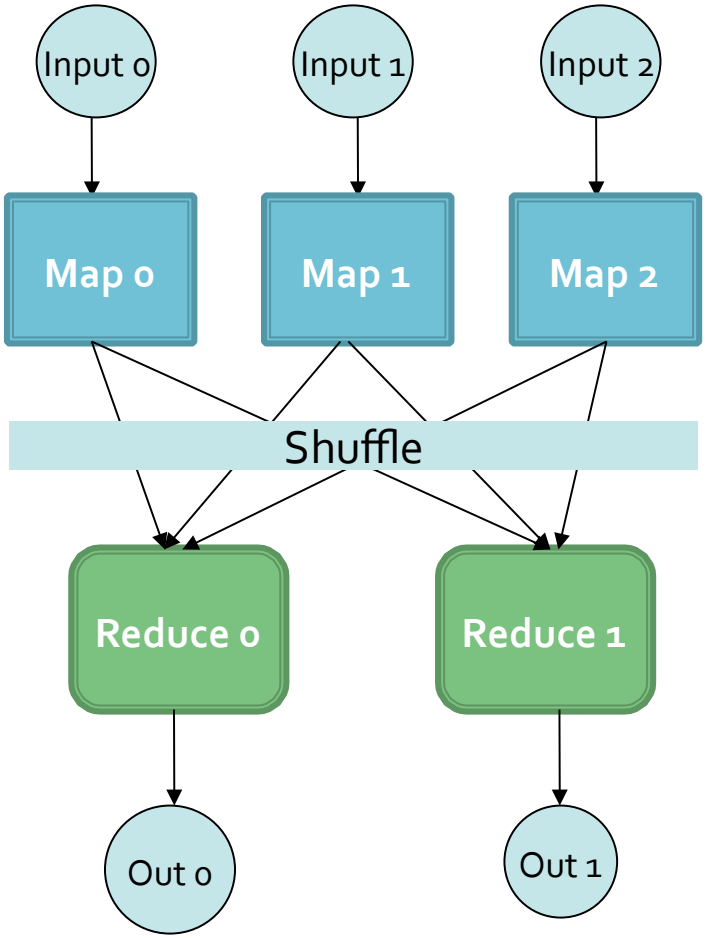
Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

Map-Reduce

- Programmer specifies:
  - Map and Reduce and input files
- **Workflow:**
  - Read inputs as a set of key-value-pairs
  - **Map** transforms input kv-pairs into a new set of k'v'-pairs
  - Sorts & Shuffles the k'v'-pairs to output nodes
  - All k'v'-pairs with a given k' are sent to the same **reduce**
  - **Reduce** processes all k'v'-pairs grouped by key into new k''v''-pairs
  - Write the resulting pairs to files
- All phases are distributed with many tasks doing the work



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks "close" to physical storage location of input data
- **Intermediate results are stored on local FS of Map and Reduce workers**
- **Output is often input to another MapReduce task**

MapReduce: Environment

MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
  - In practice this is is the bottleneck
- Handling machine **failures**
- Managing required inter-machine **communication**

Dealing with Failures

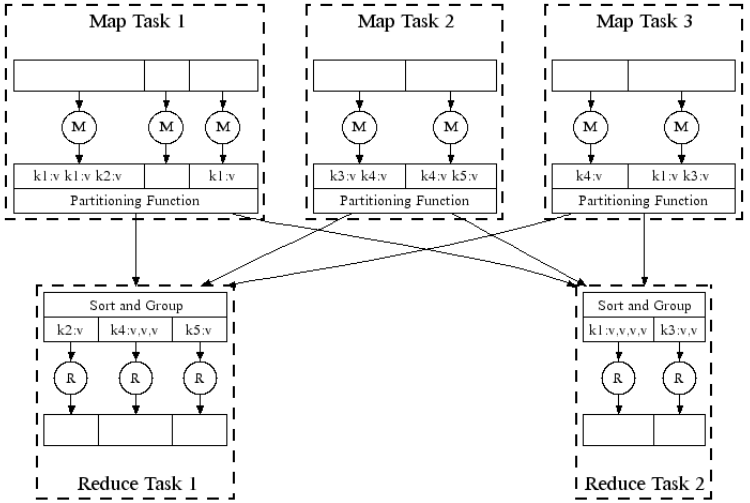
- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first "wins"
- **Effect**
  - Dramatically shortens job completion time

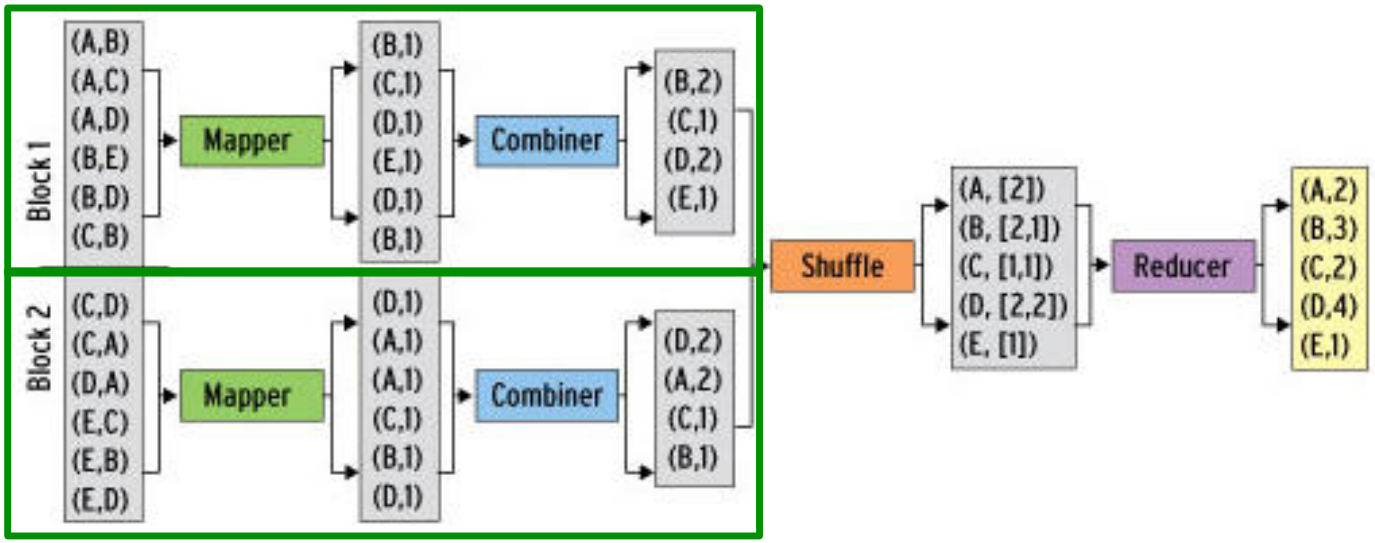
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k,v_1), (k,v_2), \ldots$ for the same key k
  - E.g., popular words in the word count example
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative



Refinement: Combiners

- **Back to our word counting example:**
  - Combiner combines the values of all keys of a single mapper (single machine):



  - Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **System uses a default partition function:**
  - **hash(key) mod *R***
- **Sometimes useful to override the hash function:**
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

Problems Suited for MapReduce

Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host

- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

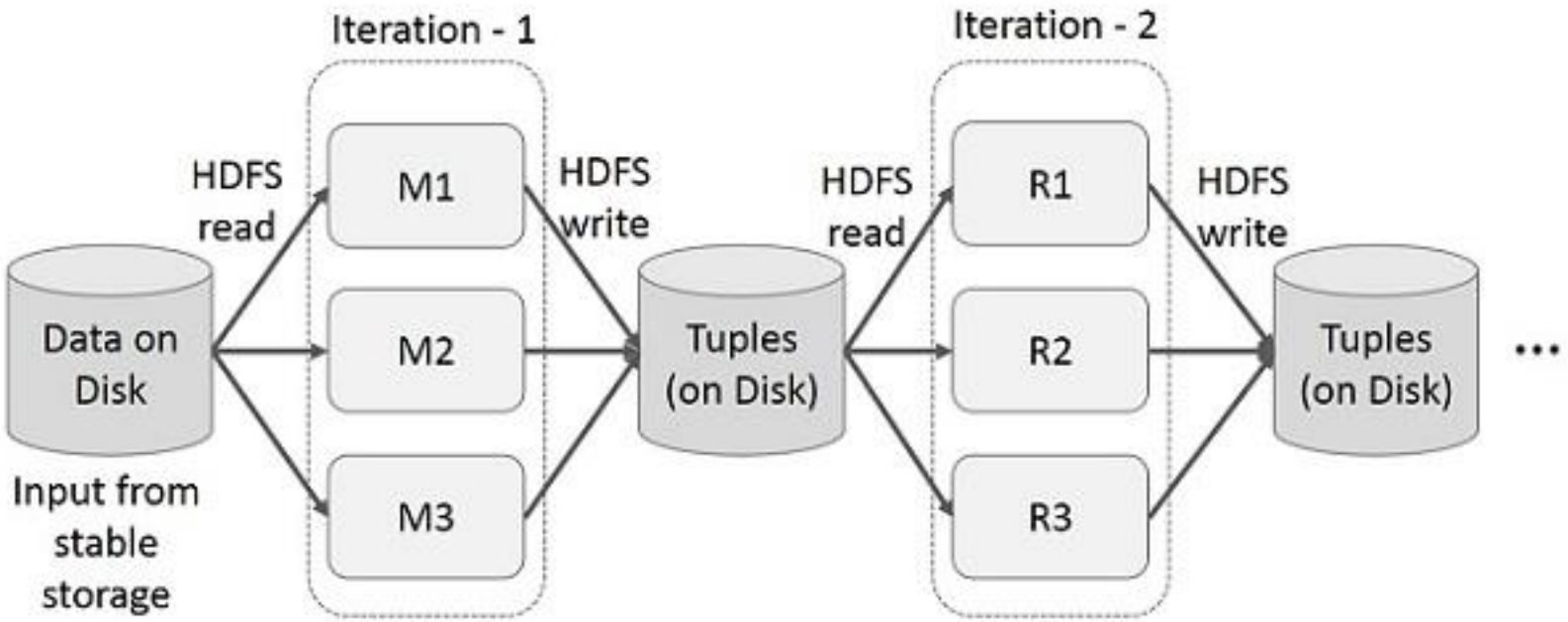
A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Spark: Extends MapReduce

Problems with MapReduce



- **MapReduce:**
  - Incurs substantial overheads due to data replication, disk I/O, and serialization

Problems with MapReduce

- **Two major limitations of MapReduce:**
  - Difficulty of programming directly in MapReduce
    - Many problems aren't easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work
- **In short, MapReduce doesn't compose well for large applications**
  - Many times, one needs to chain multiple map-reduce steps.

Data-Flow Systems

- **MapReduce uses two "ranks" of tasks:**
  One for **Map** the second for **Reduce**
  - Data flows from the first rank to the second

- **Data-Flow Systems generalize this in two ways:**
  1. Allow any number of tasks/ranks
  2. Allow functions other than Map and Reduce
  - As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs

Spark: Most Popular Data-Flow System

- **Expressive computing system, not limited to the map-reduce model**
- **Additions to MapReduce model:**
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce
- Compatible with Hadoop

Spark: Overview

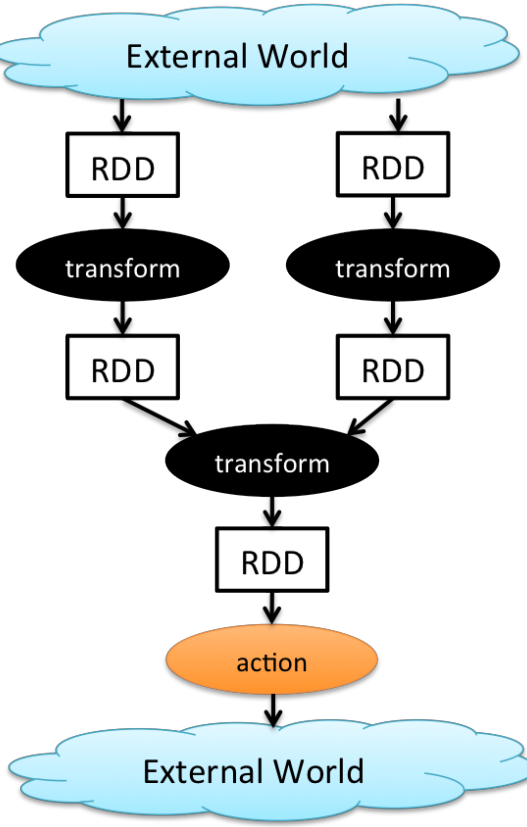
- **Key construct/idea:** Resilient Distributed Dataset (RDD)

- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support

Spark: RDD

Key concept: ***Resilient Distributed Dataset*** **(RDD)**

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Fallback to disk possible
- **RDDs** can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- **RDDs** are best suited for applications that apply the same operation to all elements of a dataset



Spark RDD Operations

- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include *map, filter, join, union, intersection, distinct*
  - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
  - Actions include *count, collect, reduce, save*
  - Actions can be applied to RDDs; actions force calculations and return values

Resilient Distributed Dataset (RDD)

Spark's key programming abstraction:

- Read-only ordered collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs
- Actions on RDDs return data to application

RDDs

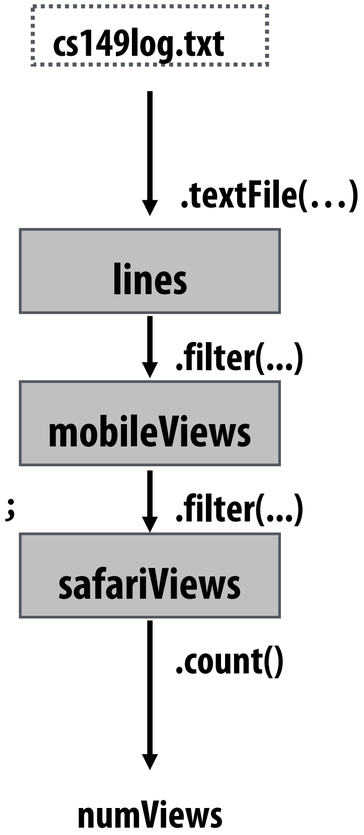
```
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
val safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
val numViews = safariViews.count();
```

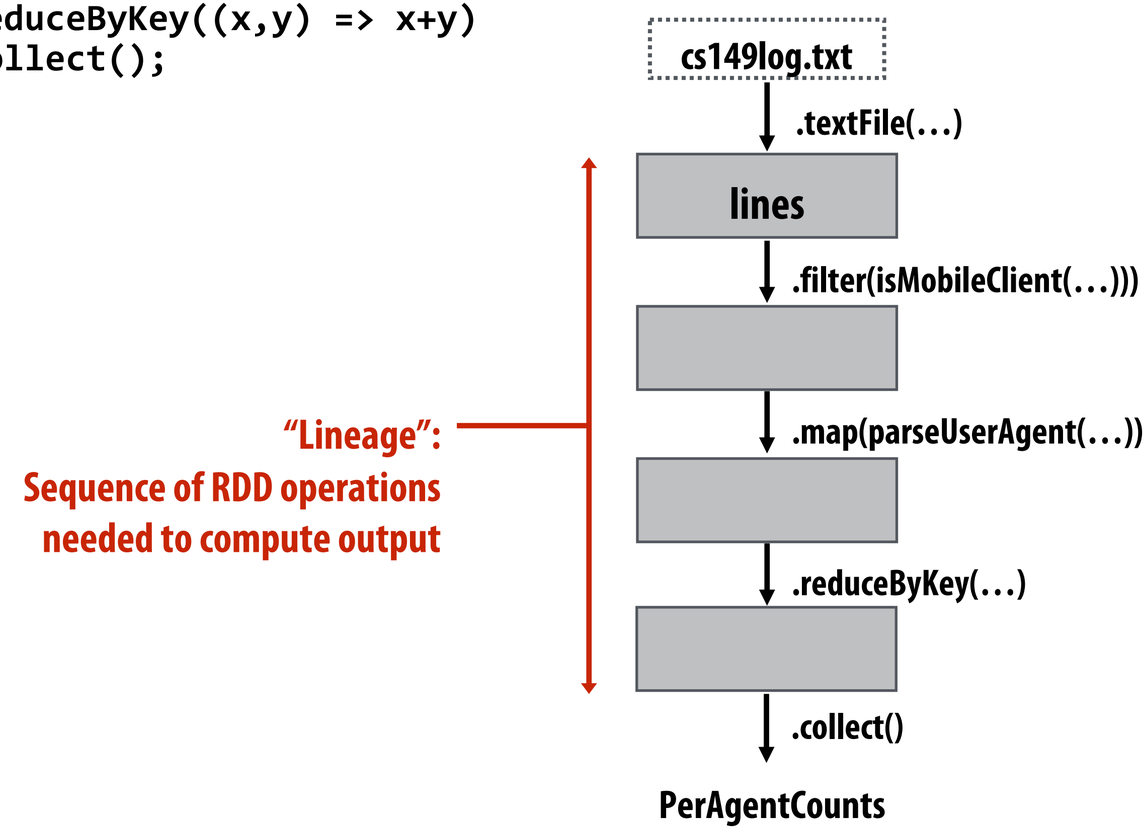
int



Repeating the MapReduce Example

```
// 1. create RDD from file system data
// 2. create RDD with only lines from mobile clients
// 3. create RDD with elements of type (String,Int) from line string
// 4. group elements by key
// 5. call provided reduction function on all keys to count views
val perAgentCounts =  spark.textFile(“hdfs://cs149log.txt”)
                           .filter(x => isMobileClient(x))
                           .map(x => (parseUserAgent(x),1));
                           .reduceByKey((x,y) => x+y)
                           .collect();
```

Array[String,int]



Another Spark Program

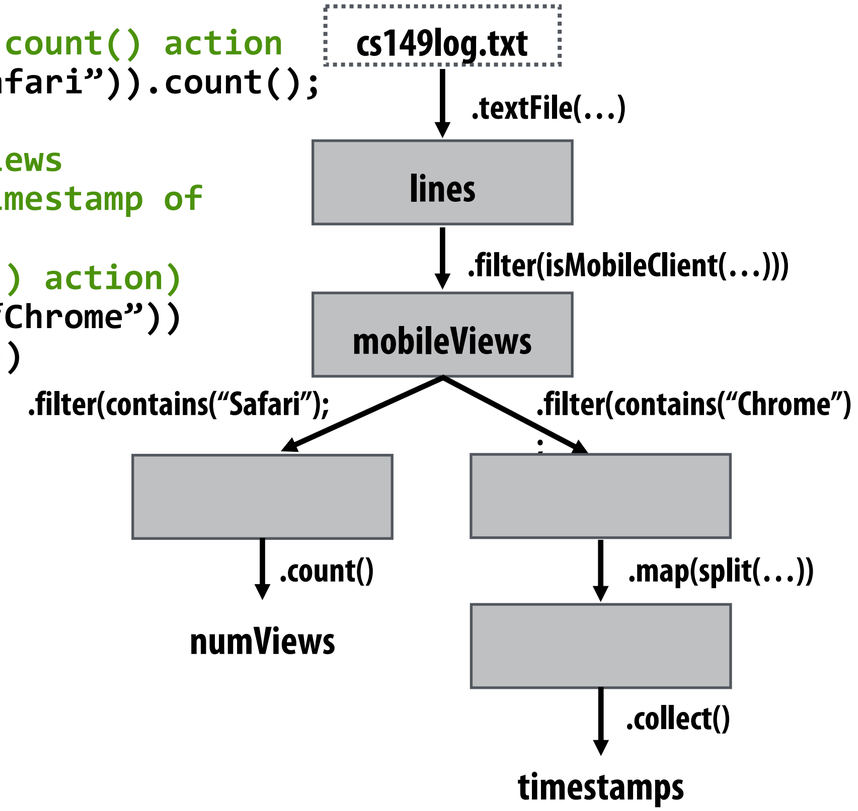
```
// create RDD from file system data
val lines = spark.textFile(“hdfs://cs149log.txt”);

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// instruct Spark runtime to try to keep mobileViews in memory
mobileViews.persist();

// create a new RDD by filtering mobileViews
// then count number of elements in new RDD via count() action
val numViews = mobileViews.filter(_.contains(“Safari”)).count();

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of
//    page view
// 3. convert RDD to a scalar sequence (collect() action)
val timestamps = mobileViews.filter(_.contains(“Chrome”))
                            .map(_.split(“ ”)(0))
                            .collect();
```



RDD transformations and actions

Transformations: (data parallel operators taking an input RDD to a new RDD)

$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]

Actions: (provide data back to the "host" application)

$count()$	:	RDD[T] ⇒ Long
$collect()$	:	RDD[T] ⇒ Seq[T]
$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed

File: purplecow.txt

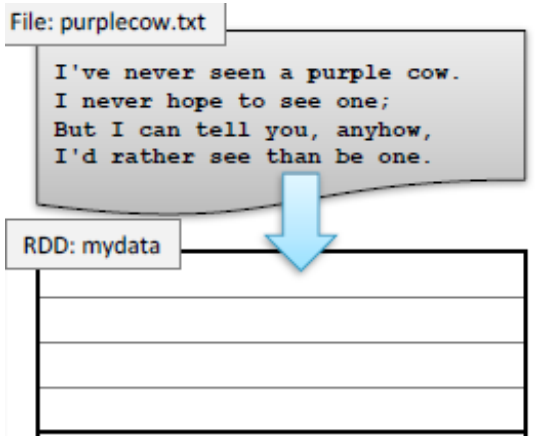
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

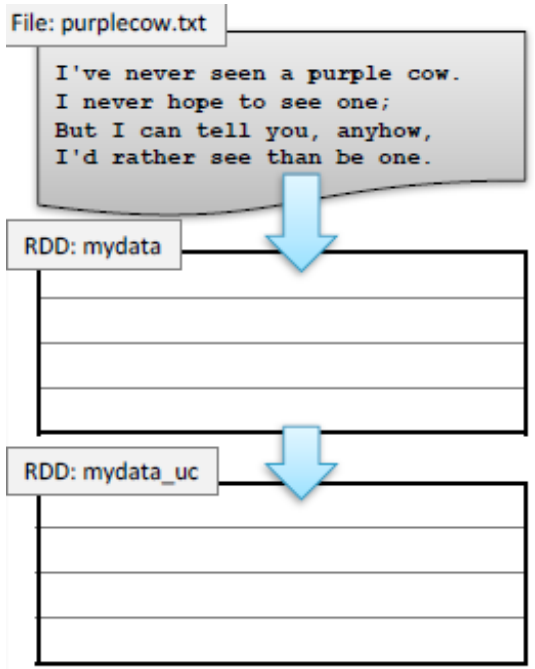
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

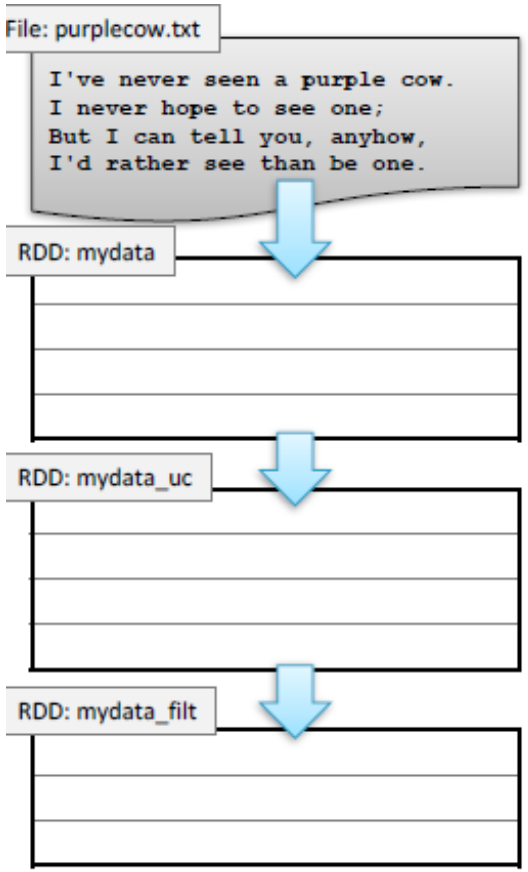
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

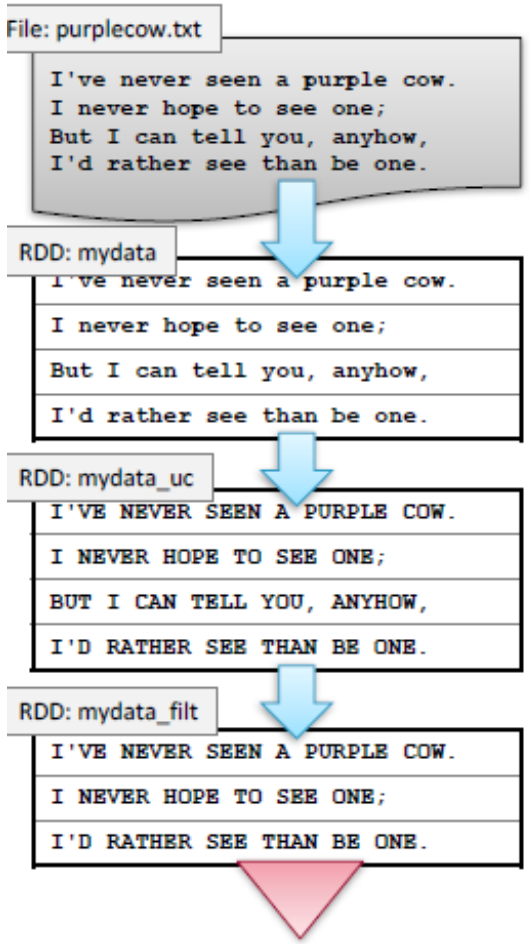
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution of RDDs (5)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



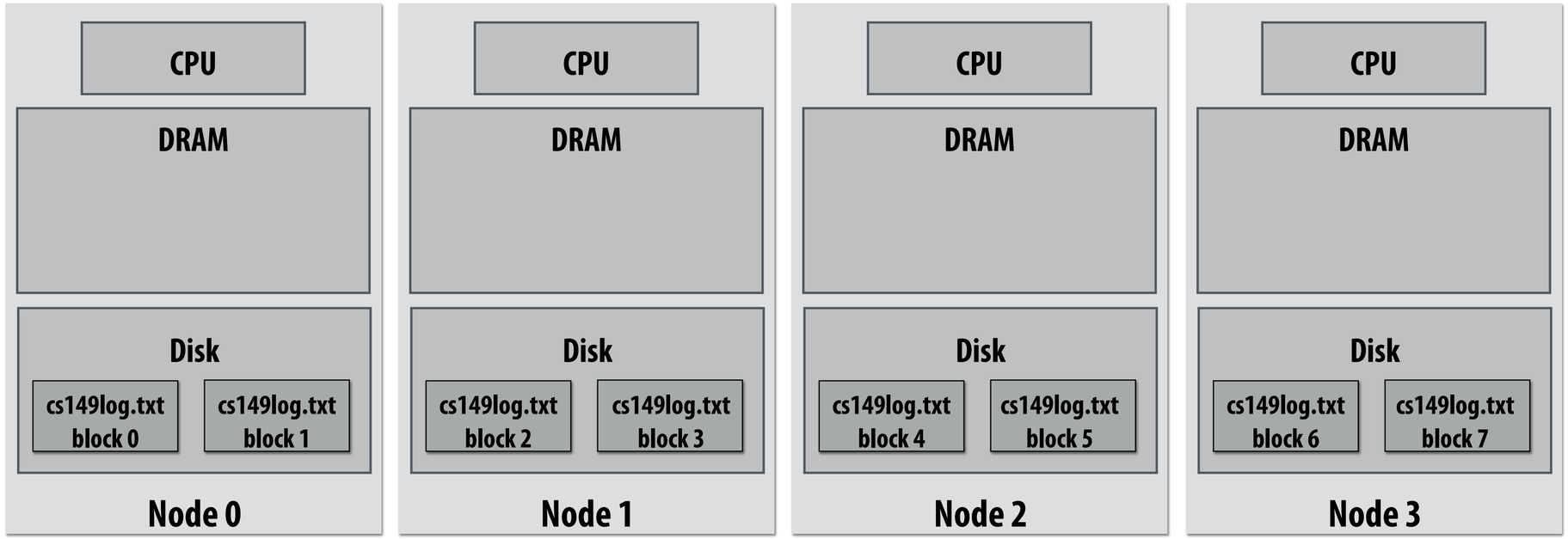
Output Action "triggers" computation, pull model

How do we implement RDDs?

In particular, how should they be stored?

```
val lines = spark.textFile(“hdfs://cs149log.txt”);
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Question: should we think of RDD’s like arrays?



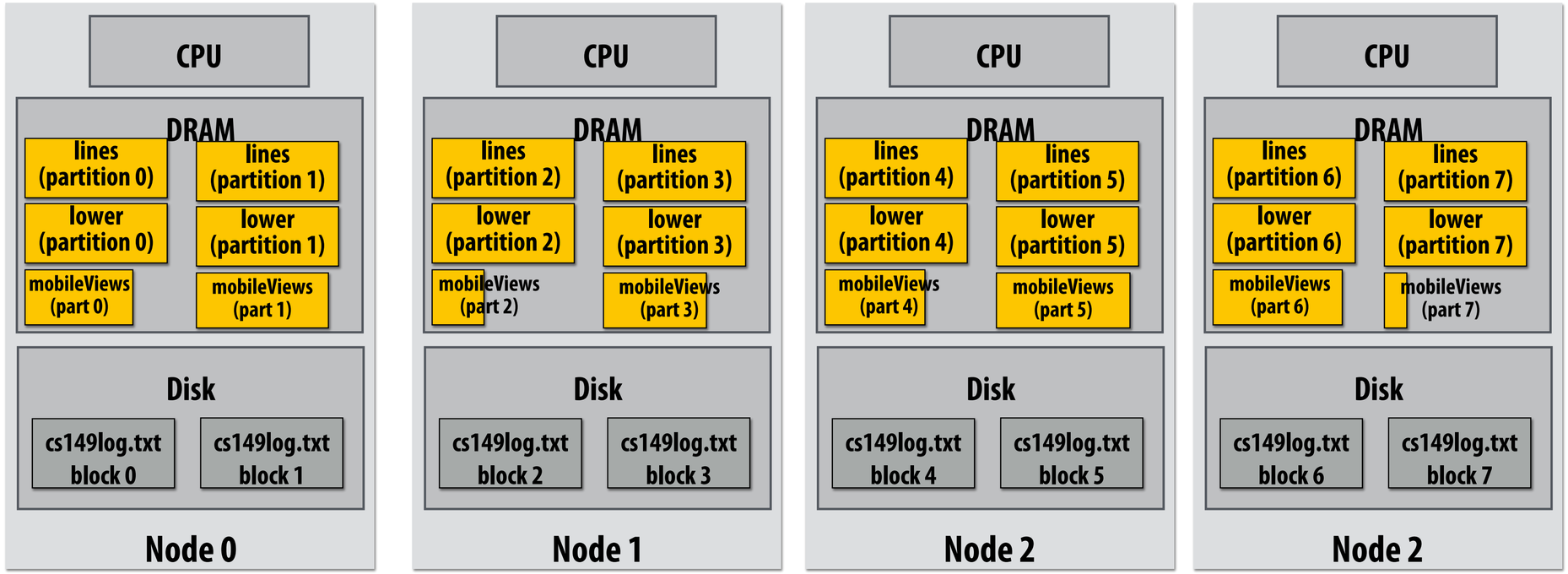
How do we implement RDDs?

In particular, how should they be stored?

Parallel Performance = Parallelism + Locality

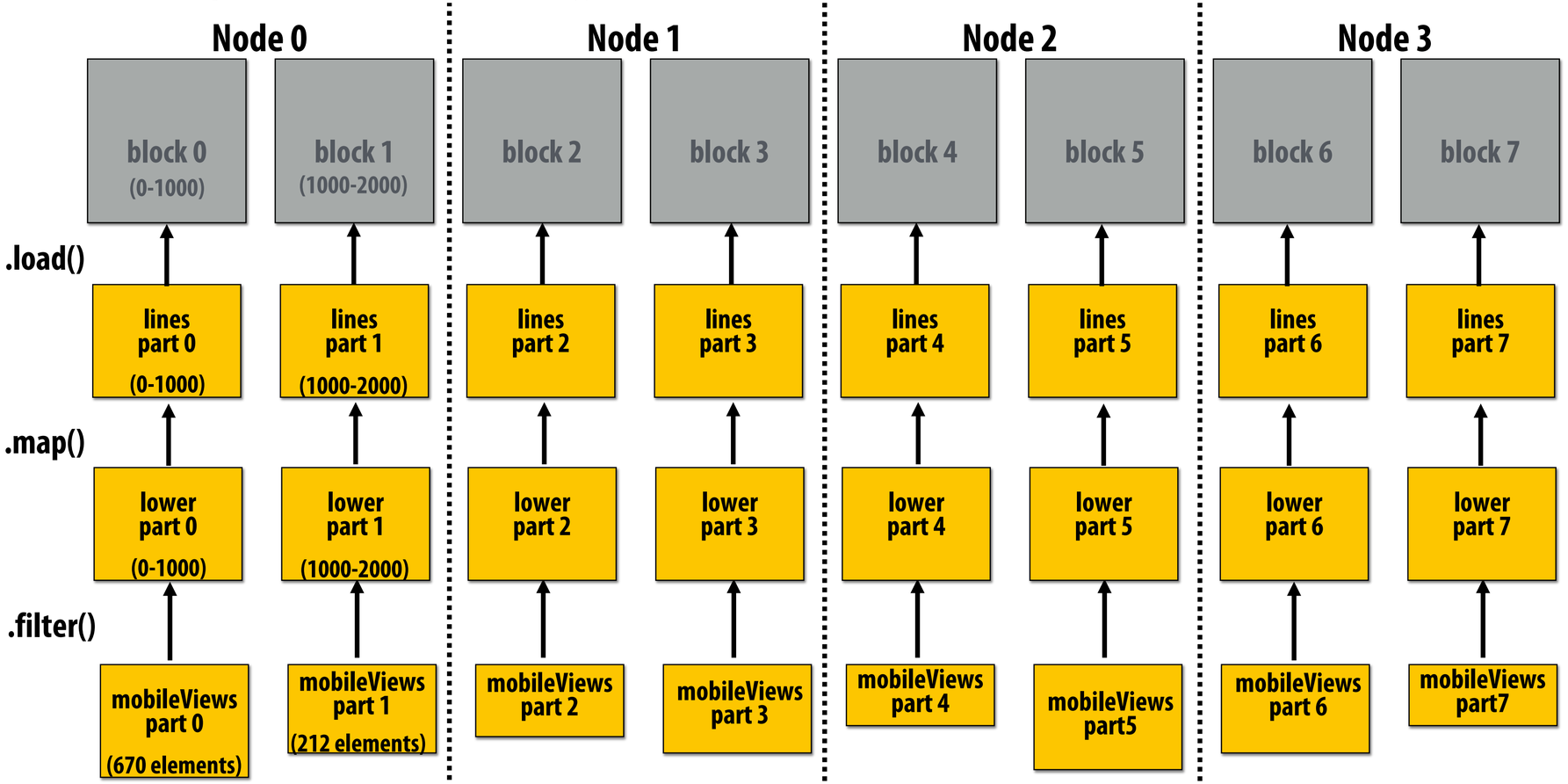
```
val lines = spark.textFile(“hdfs://cs149log.txt”);
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

In-memory representation would be huge! (larger than original file on disk)



RDD partitioning and dependencies

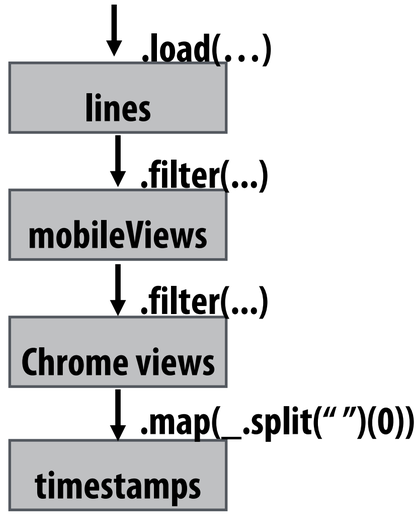
```
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```



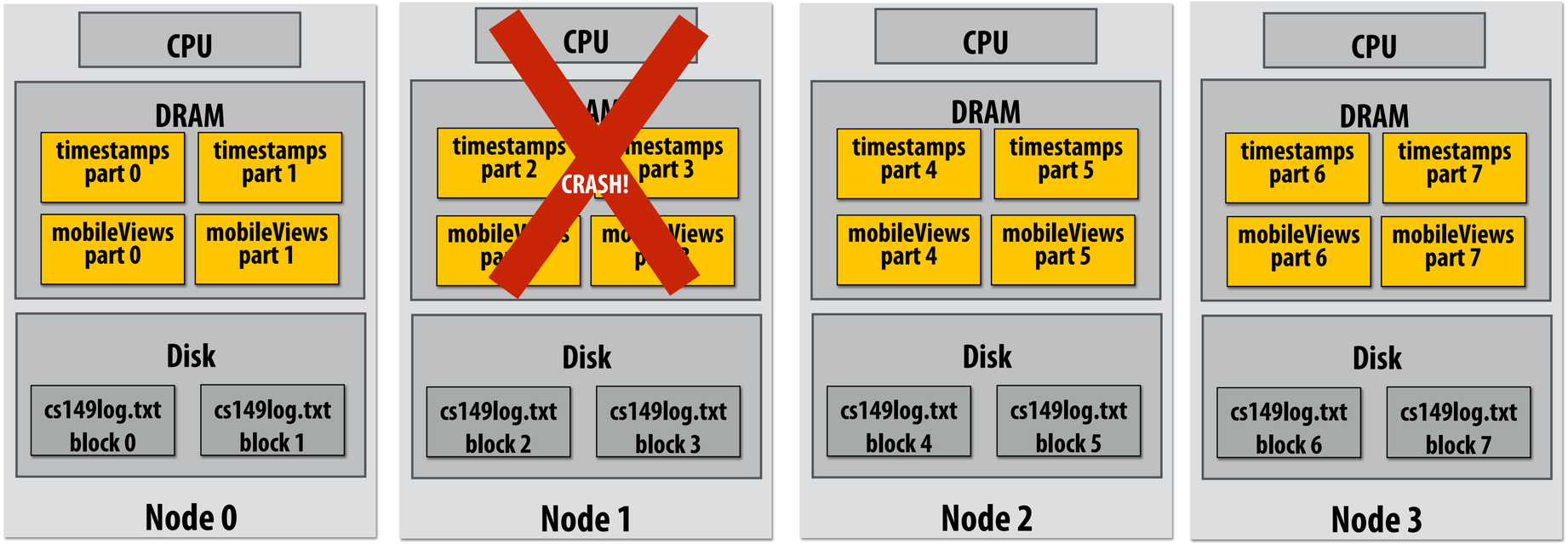
Black lines show dependencies between RDD partitions

Upon Node Failure: Recompute Lost RDD Partitions from Lineage

```
val lines       = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps  = mobileView.filter(_.contains("Chrome"))
                            .map(_.split(" ")(0));
```



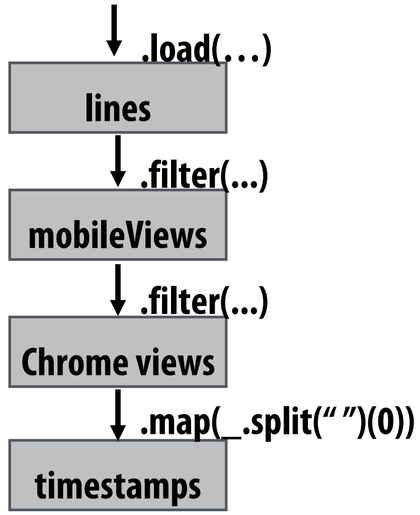
Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.



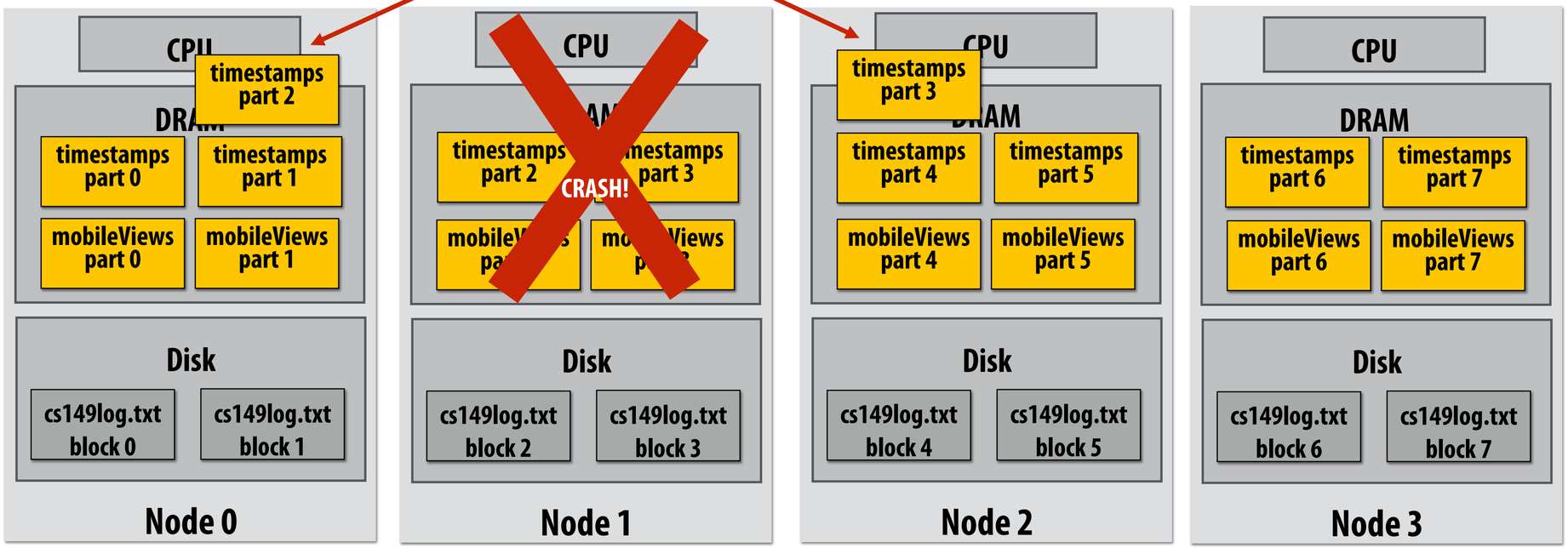
Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes

Upon Node Failure: Recompute Lost RDD Partitions from Lineage

```
val lines       = spark.textFile(“hdfs://cs149log.txt”);
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps  = mobileView.filter(_.contains(“Chrome”))
                            .map(_.split(“ ”)(0));
```



Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps



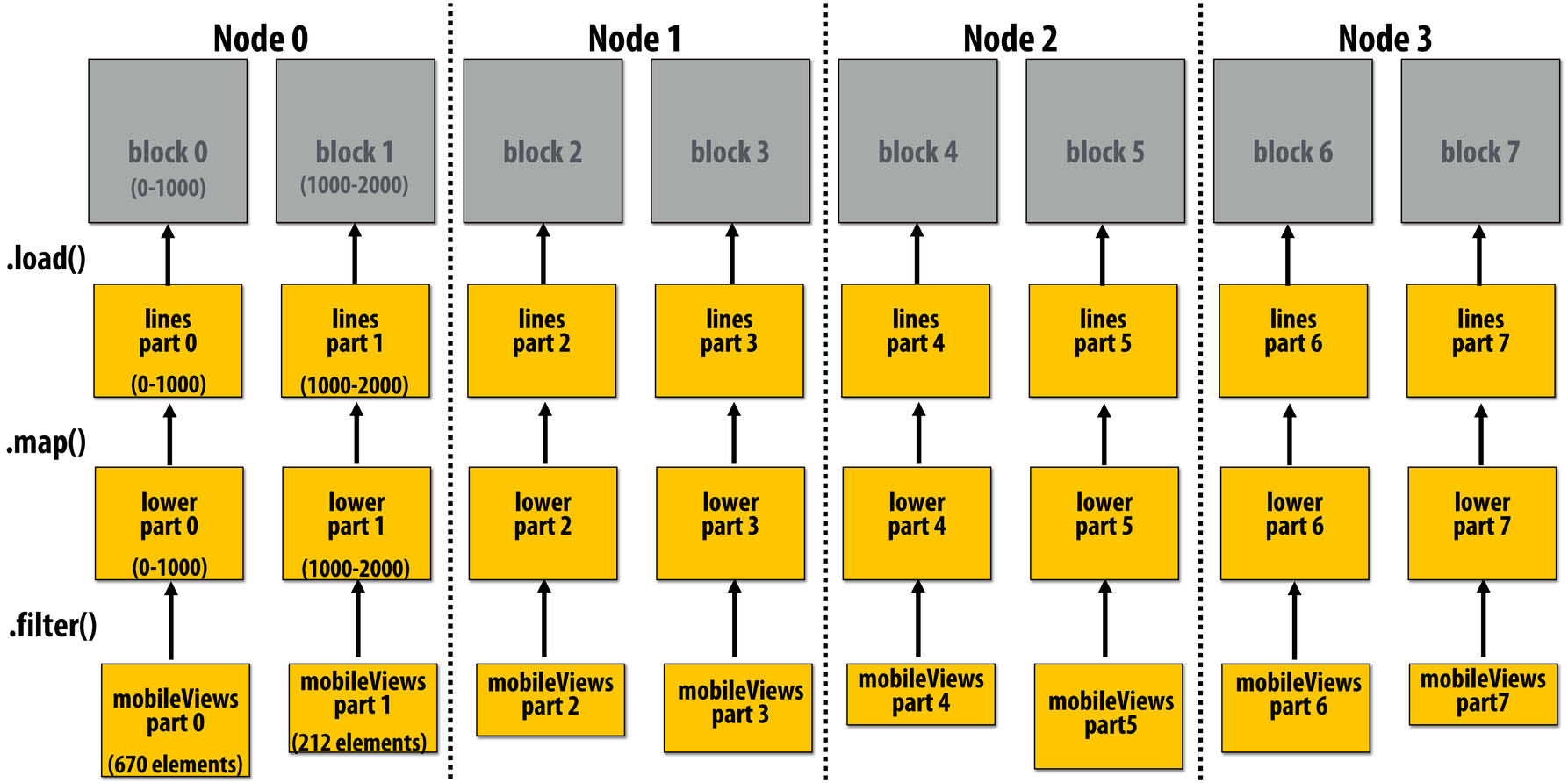
Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes

Narrow dependencies

```
val lines = spark.textFile(“hdfs://cs149log.txt”);
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

“Narrow dependencies” = each partition of parent RDD referenced by at most one child RDD partition

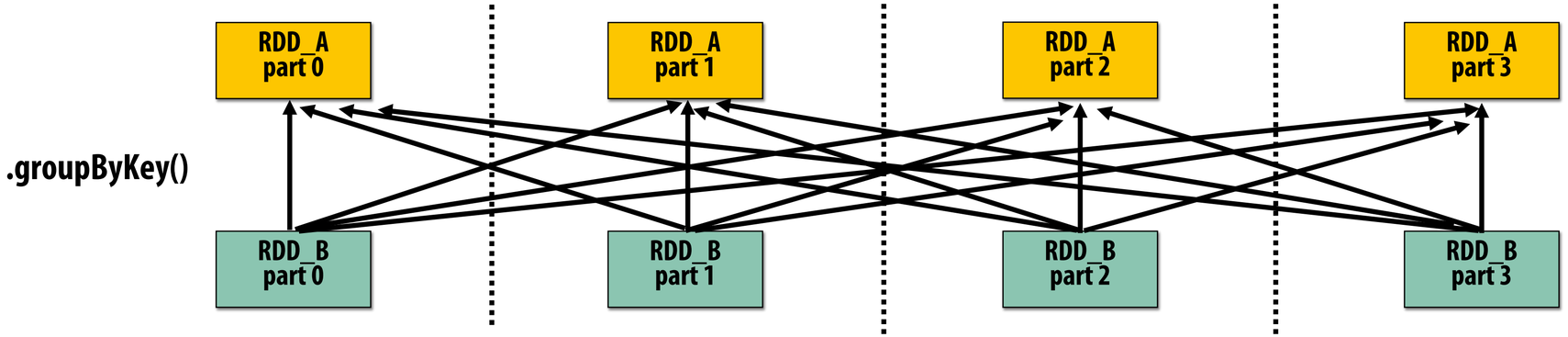
- Allows for fusing of operations (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster (communication of one int at end to perform count() reduction)



Wide dependencies

groupByKey: RDD[(K,V)] ⟶ RDD[(K,Seq[V])]

"Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key."



Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

Challenges:

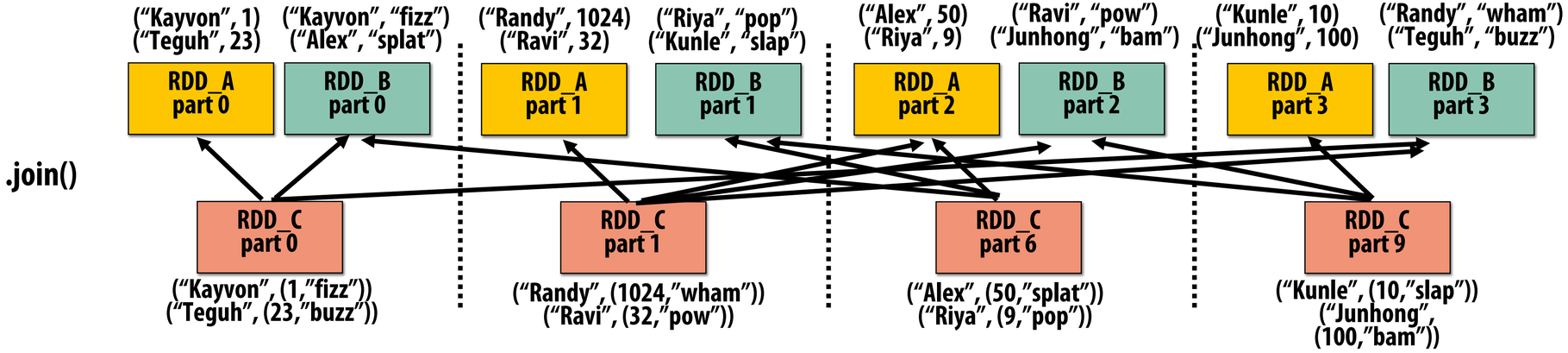
- Must compute all of RDD_A before computing RDD_B
  - Example: groupByKey() may induce all-to-all communication as shown above
- May trigger significant recomputation of ancestor lineage upon node failure (I will address resilience in a few slides)

Cost of operations depends on partitioning

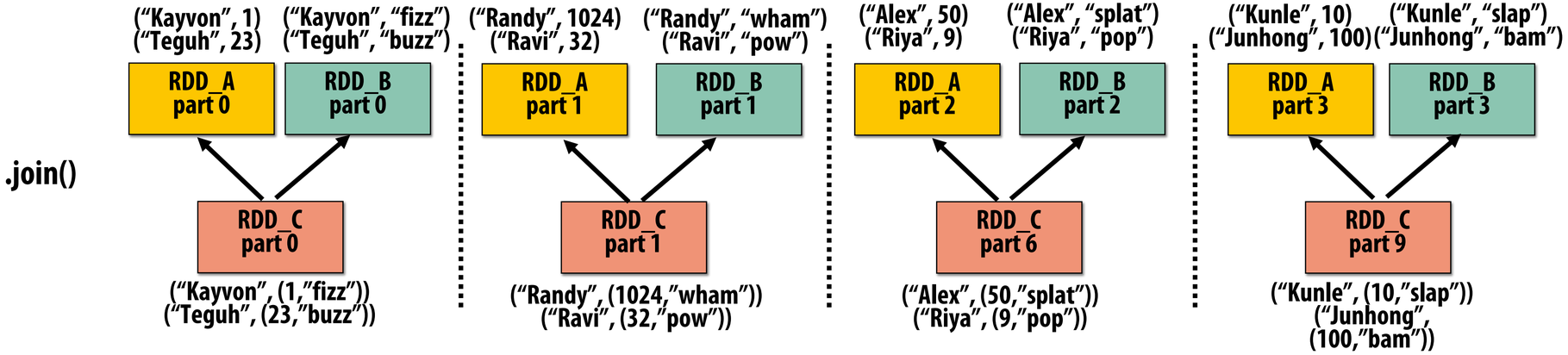
join: RDD[(K,V)], RDD[(K,W)] → RDD[(K,(V,W))]

Assume data in RDD_A and RDD_B are partitioned by key: hash username to partition id

RDD_A and RDD_B have different hash partitions: join creates wide dependencies



RDD_A and RDD_B have same hash partition: join only creates narrow dependencies



Higher-Level API: DataFrame & Dataset

- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection)

Both built on Spark SQL engine. Both can be converted back to an RDD.

Useful Libraries for Spark

- Spark SQL
- Spark Streaming – **stream processing of live datastreams**
- MLlib – **scalable machine learning**
- GraphX – **graph manipulation**
  - Extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge

Spark vs. Hadoop MapReduce

- **Performance:** Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it **often needs lots of memory to perform well**; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: **Spark is easier to program** (higher-level APIs)
- Data processing: **Spark more general**