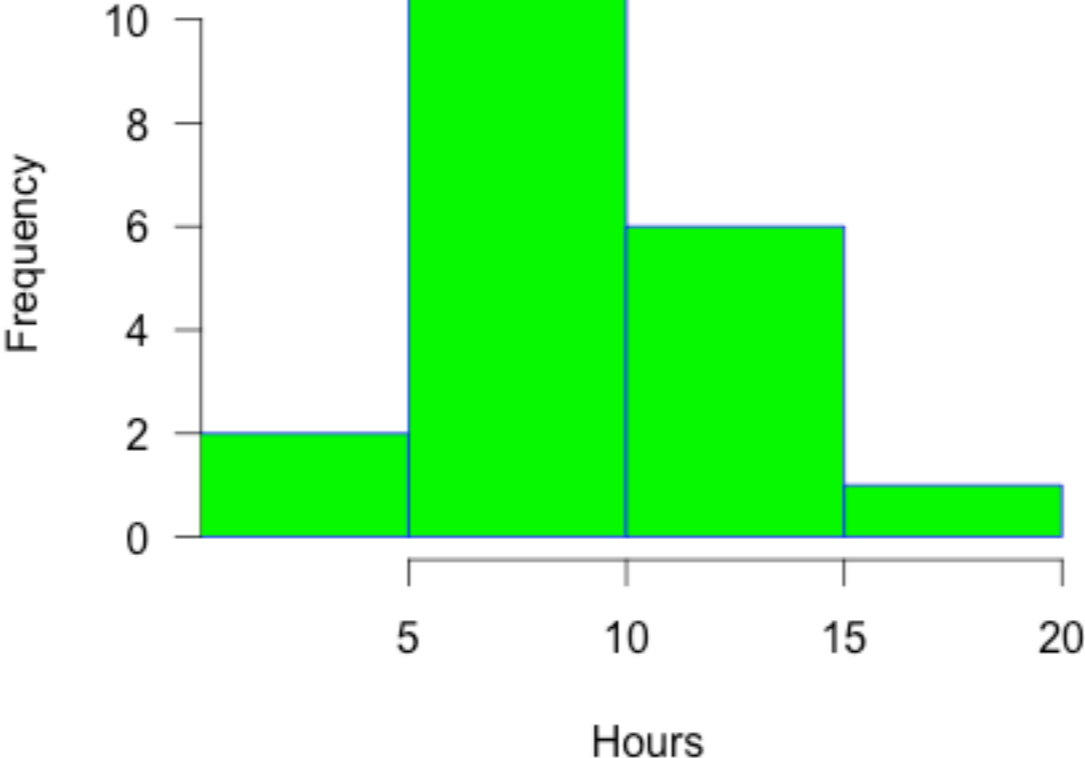


Comp 311

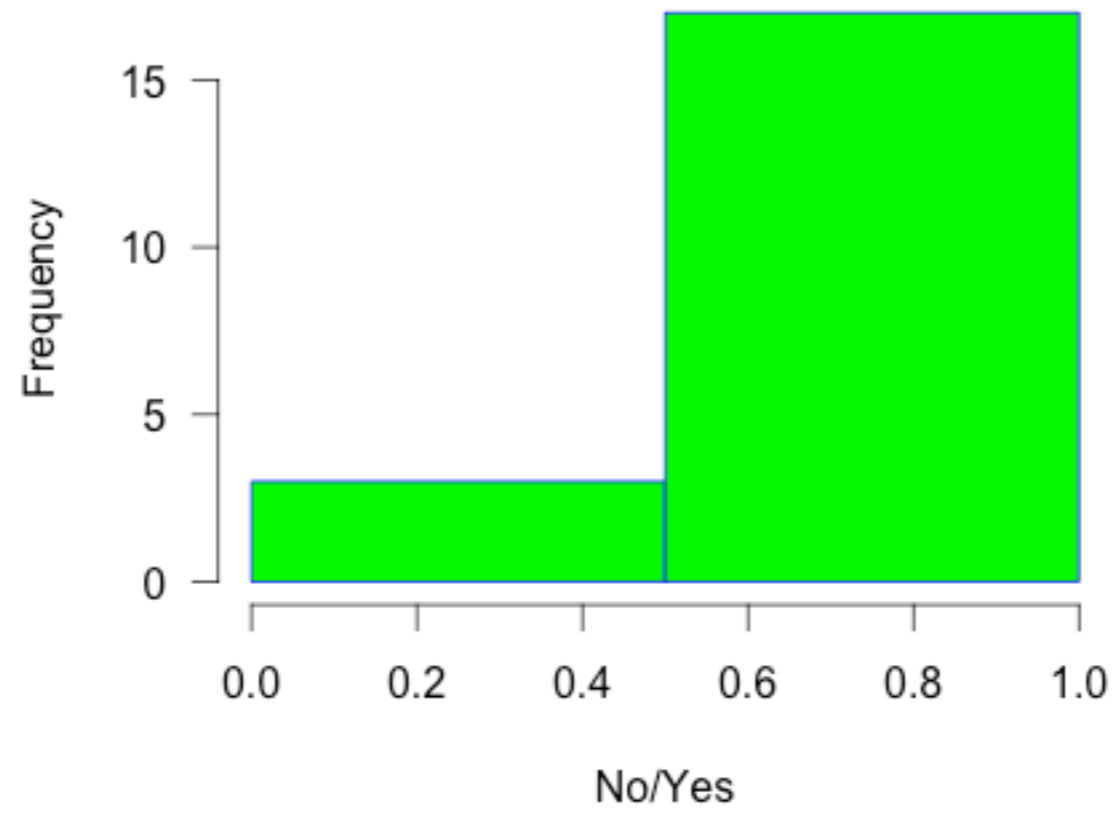
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC

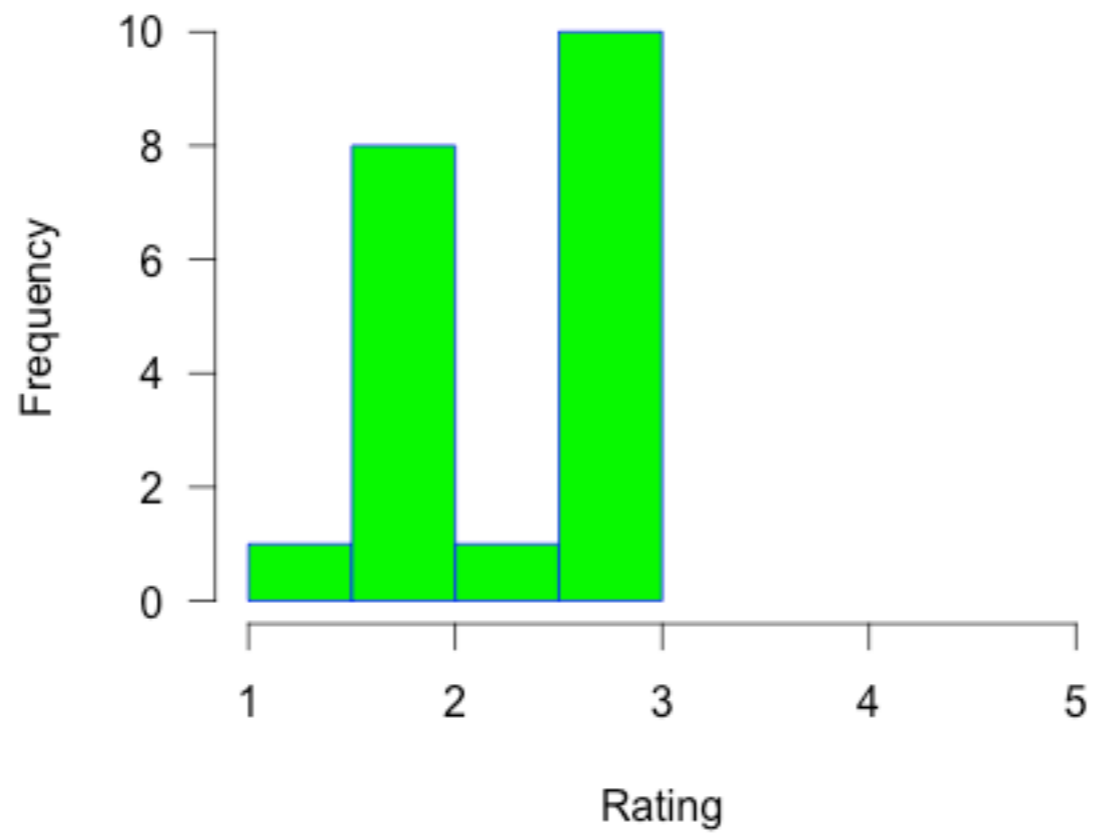
Homework 5 Hours Spent



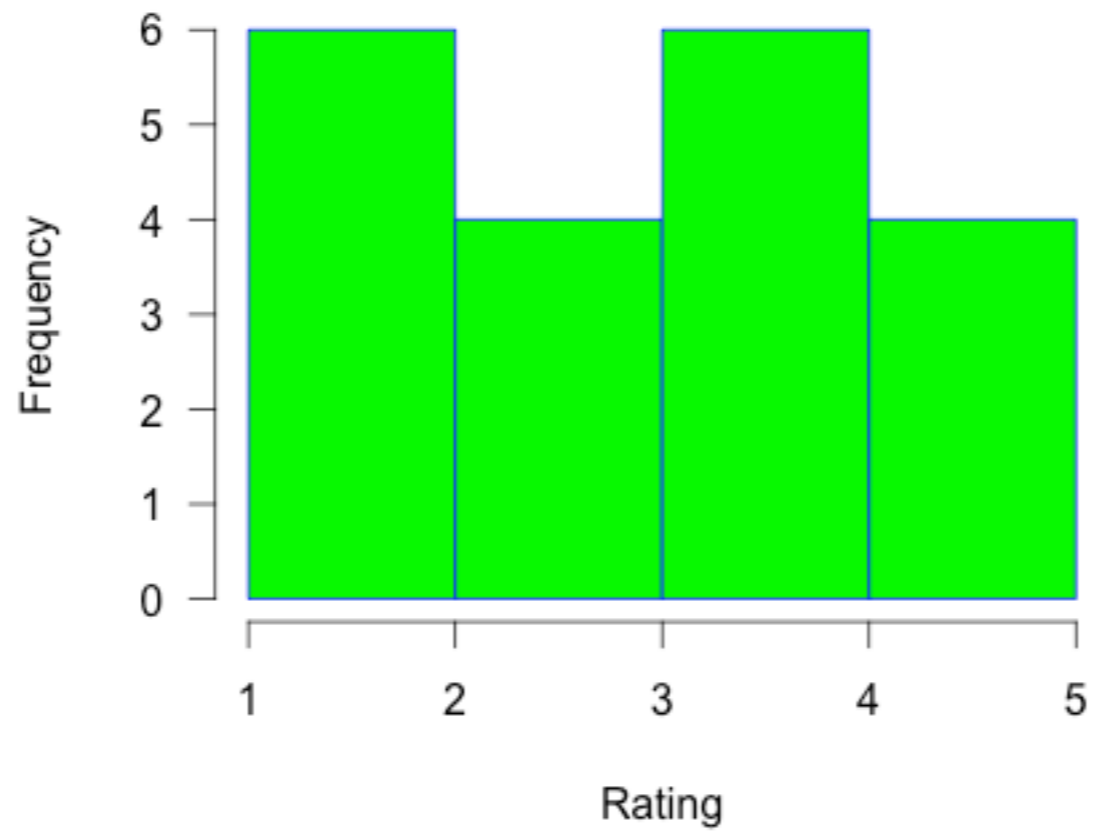
Homework 5 Complete



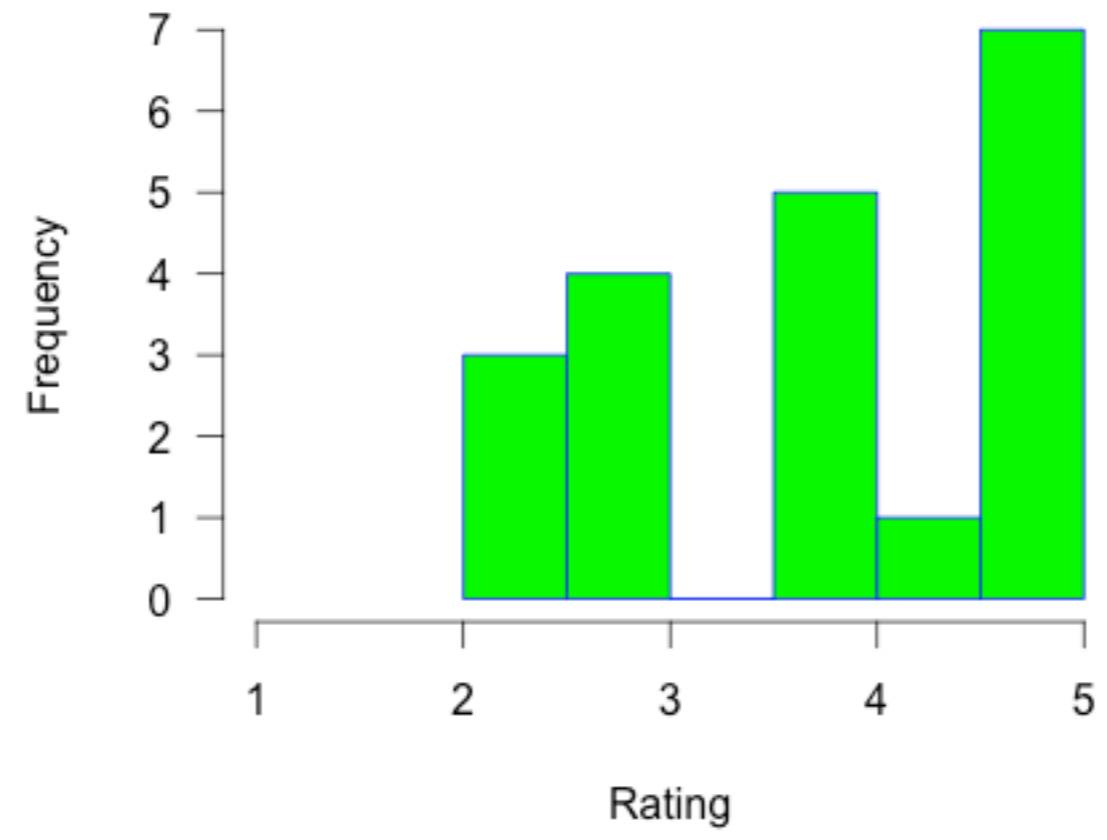
Homework 5 Workload



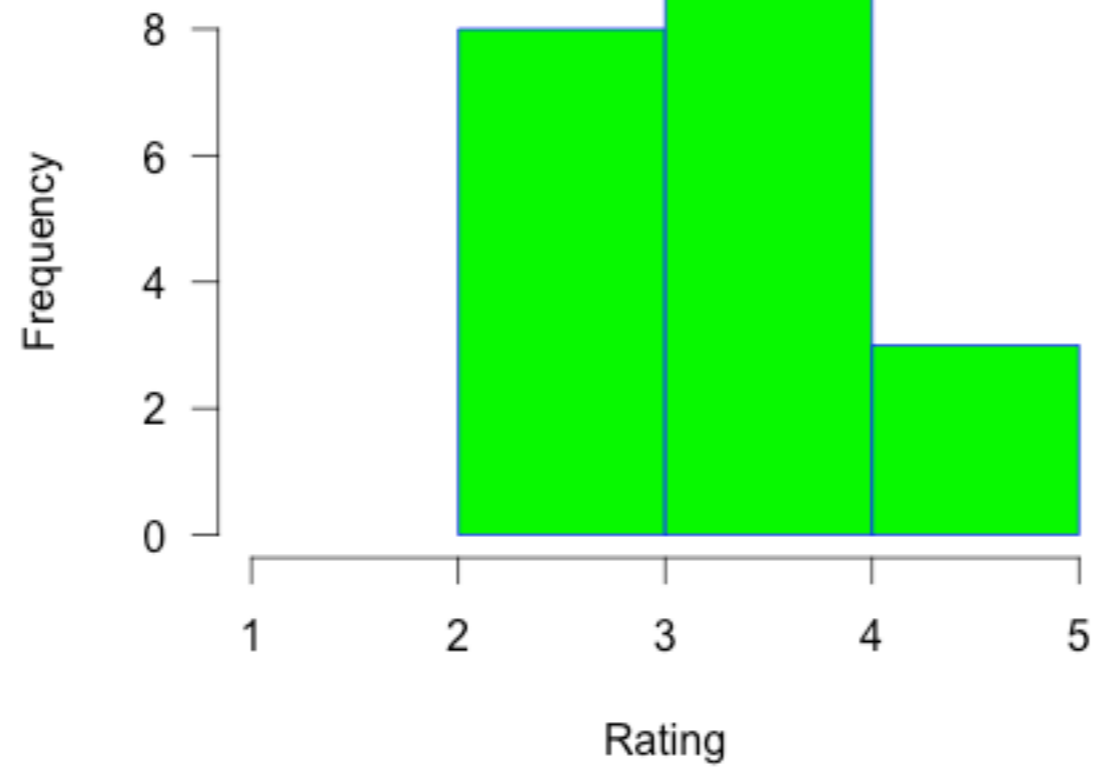
Homework 5 Helpful



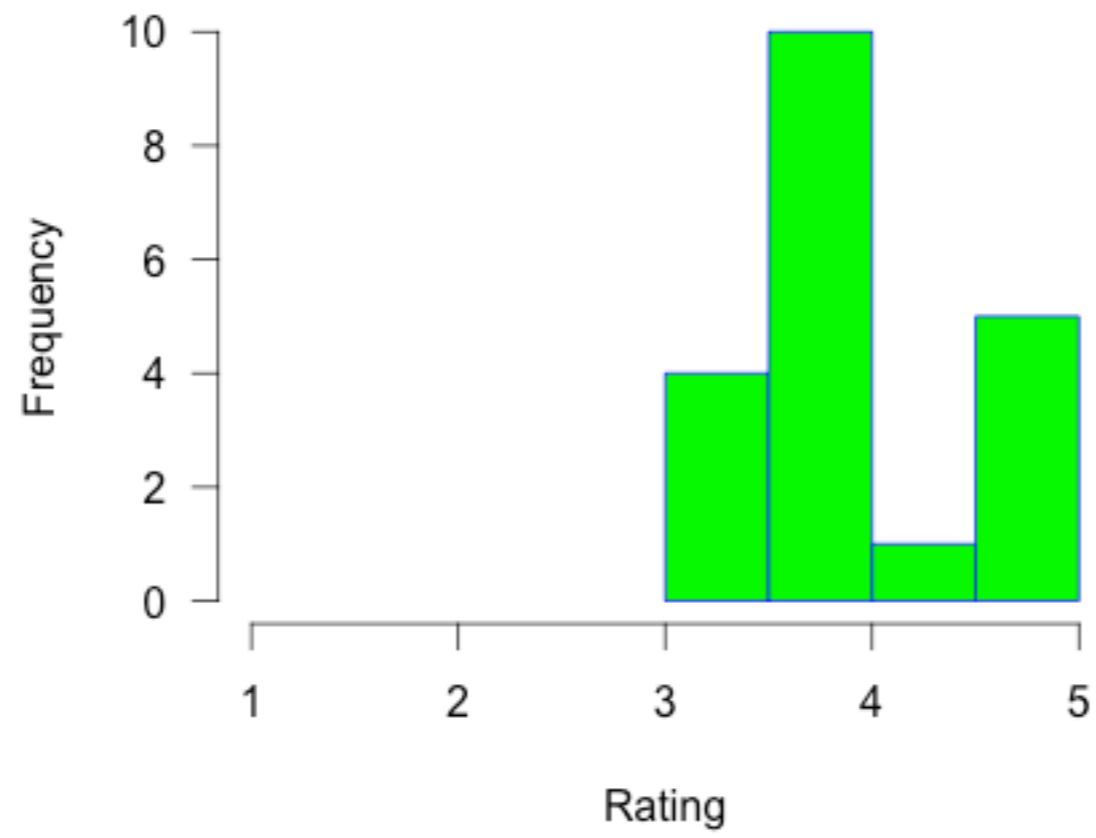
Homework 5 Enjoyable



Lectures Easy to Follow



Class Enjoyable



Functional Programming and Large Scale Data Analytics

Large Scale Data Analytics

- Many trends have resulted in a dramatic rise in the amount of data available for computation:
 - e-commerce
 - social networking
 - mobile phones
 - etc.

Large Scale Data Analytics

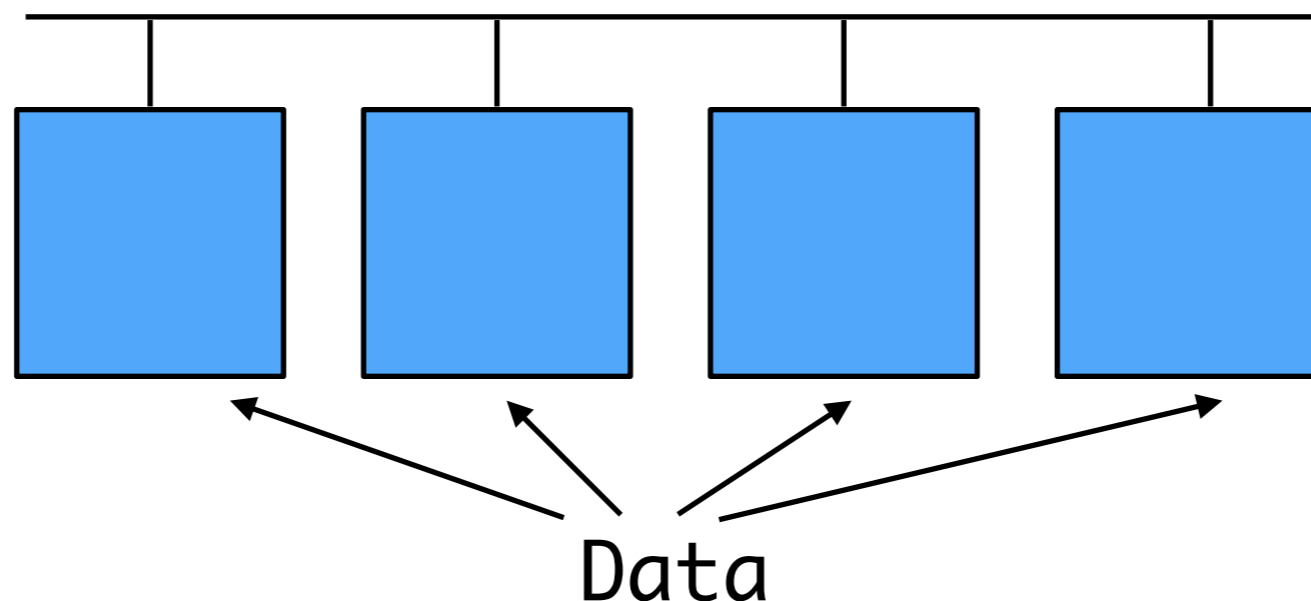
- Significant value can be gleaned from quick processing of large datasets
 - Real-time navigation adjustments
 - Targeted online advertising
 - Customized medical diagnosis
 - Retail recommendations
 - More relevant news and social feeds

Large Scala Data Analytics

- AT&T processes roughly 30 petabytes per day through its telecommunications network
- Google processed roughly 24 petabytes per day in 2009
- Facebook, Amazon, Twitter, etc, have comparable throughputs
- Two Sigma maintains over 100 teraflops of private computing power continuously computing over 11 petabytes of quantitative data
- By comparison, the IBM Watson knowledge base stored roughly 4 terabytes of data when winning at Jeopardy

Challenges in Computing Large-Scale Streaming Data

- Continuously streaming data needs to be processed at least as fast as it is accumulated, or we will never catch up
- The bottleneck in processing very large data sets has been dominated for many years by the speed of disk access
- More processors accessing more disks enables faster processing



Cloud Computing

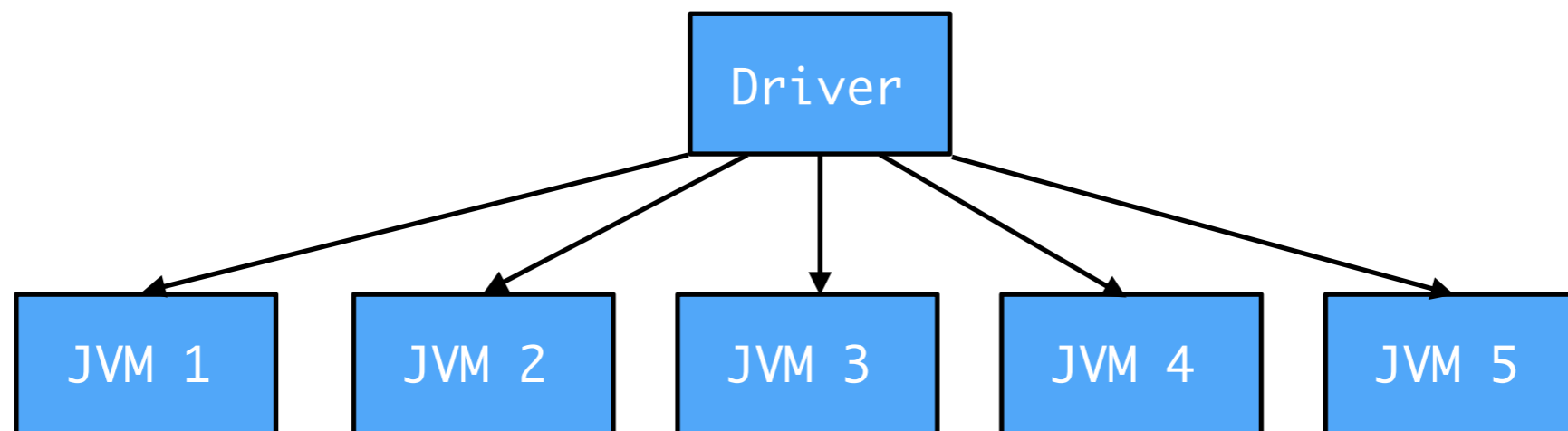
- Computing, storage, and communication at pennies per hour
- No premium to scale:
 - 1000 computers @ 1 hour = 1 computer @ 1000 hours
- Provides the illusion of infinite scalability to cloud user
 - Use as many computers as you can afford
- Leading examples: Amazon Web Services (AWS), Google App Engine, Microsoft Azure

Cloud Computing

- Economies of scale have pushed down datacenter costs by factors of 3-8
 - Traditional datacenters utilized 10% - 20% of their machines
 - Cloud computing services are far more economical
- But how do we extract portable and scalable parallelism from our programs?
 - One solution: Take advantage of functional programming to express simple parallelism easily

Cluster Computing Frameworks

- Enabled writing of parallel computations using functional operators, without worrying about distribution and fault tolerance



MapReduce

- Load a large data set from disk on to multiple machines
- Map a function over that data to return key value pairs
- Shuffle results so that pairs with the same keys are brought together
- Reduce to one value for each key
- Write result to disk

MapReduce

- Computations that involve a sequence of iterations of map/reduce operations pay a heavy price:
 - Each iteration must read from and write to disk

Iterative Map/Reduce Schedulers

- Users started to realize that a much larger class of algorithms could be expressed as an iteration of MapReduce operations
 - Many machine learning algorithms fall into this category
- Tools started to emerge to enable easy expression of map/reduce operations along with smart scheduling

Apache Spark

- Cache results of map/reduce operations so they can be used on subsequent iterations
- 10-100 times faster than MapReduce for many applications

Resilient Distributed Datasets

- Fault-Tolerant parallel data structures
- Enable users to:
 - Persist intermediate results in memory
 - Control partitioning to optimize data placement
 - Manipulate data with many available operators

Resilient Distributed Datasets

- Immutable
- Operators are *coarse-grained*: map, filter, join, etc.
- Allows for efficient fault tolerance by logging the operations applied to build a dataset rather than the actual dataset

Spark and Resilient Distributed Datasets

- Partitioned across the many machines of a cluster
- Created by:
 - Reading data from storage
 - Performing transformations on other RDDs

Resilient Distributed Datasets

- Stores information as to how it was derived from other datasets
- Able to compute its partitions from data on disk
- Impossible to reference an RDD that cannot be reconstructed after a failure

Resilient Distributed Datasets

- Persistence
 - A program:
 - Indicates which RDDs will be reused
 - Chooses a storage strategy for each RDD

Resilient Distributed Datasets

- Partitioning
 - A program:
 - Asks that RDDs are partitioned across machines based on a key in each record
 - Useful for ensuring that two datasets can be joined efficiently

The RDD API

- RDDs defined through *transformations* on data on disk
 - map, filter, etc.
- RDDs are used in *actions*:
 - Operations that return a value or export data to disk
 - count, collect, save
 - No work is done on the cluster until an action forces it

The RDD API

- RDDs have a `persist` method
 - Indicates that an RDD will be used in subsequent operations
 - Implementation attempts to keep persisted RDDs in memory of the machines in a cluster
 - Spills to disk gracefully

Creating an RDD

```
val lines = spark.textFile("hdfs://...")  
val errors = lines.filter(_.startsWith("ERROR"))  
errors.persist()
```

No work has been done yet on the cluster.

Creating an RDD

```
val lines = spark.textFile("hdfs://...")  
val errors = lines.filter(_.startsWith("ERROR"))  
errors.persist()
```

The RDD records the transformations performed to compute it from disk, enabling recomputation after a failure.

Performing an Action on an RDD

```
errors.count()
```

An action will force computation of the RDD.

Performing an Action on an RDD

```
errors.count()
```

Because errors is now stored in memory, subsequent computations involving errors will be much faster.

Performing an Action on an RDD

```
errors.count()
```

*Note that the lines RDD is never stored in memory.
(Why is this behavior desirable?)*

Performing an Action on an RDD

```
errors.filter(_.contains("MySQL"))
```

*New RDDs can be computed via
transformations on existing RDDs.*

Performing an Action on an RDD

```
errors.filter(_.contains("MySQL")).count()
```

Again, computation is not performed until an action forces it.

Performing an Action on an RDD

```
errors.filter(_.contains("HDFS"))  
  .map(_.split('\t')(3))  
  .collect()
```

*Splits a line into an array of elements,
according to occurrences of the tab character.*

Performing an Action on an RDD

```
errors.filter(_.contains("HDFS"))  
  .map(_.split('\t')(3))  
  .collect()
```

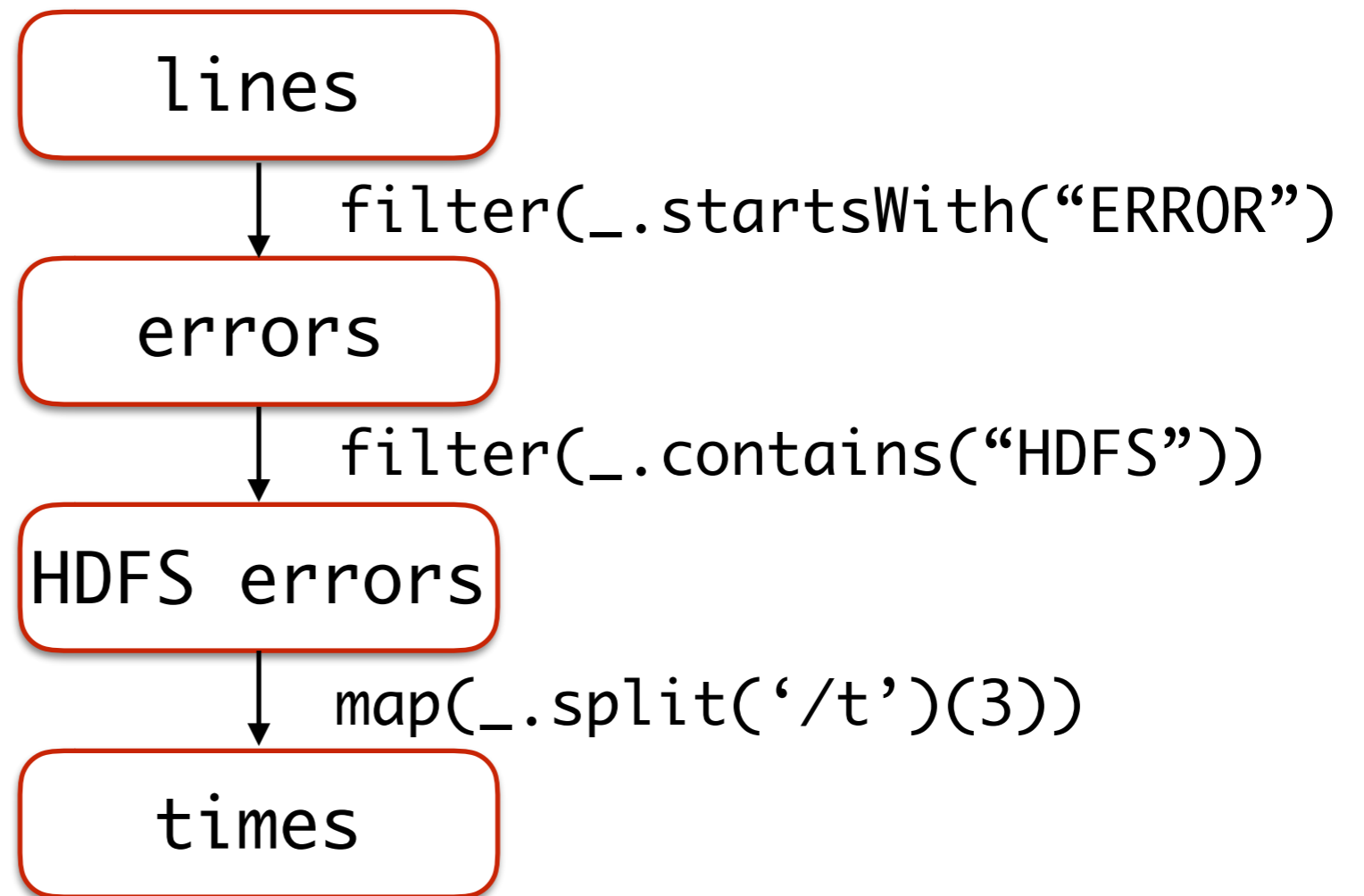
*Accesses the fourth element of each array.
(In this case, let's say that the fourth element of
the log lines stores time.)*

Performing an Action on an RDD

```
val times =  
  for (x <- errors if x contains "HDFS")  
    yield x.split('\t')(3)  
  
times.collect()
```

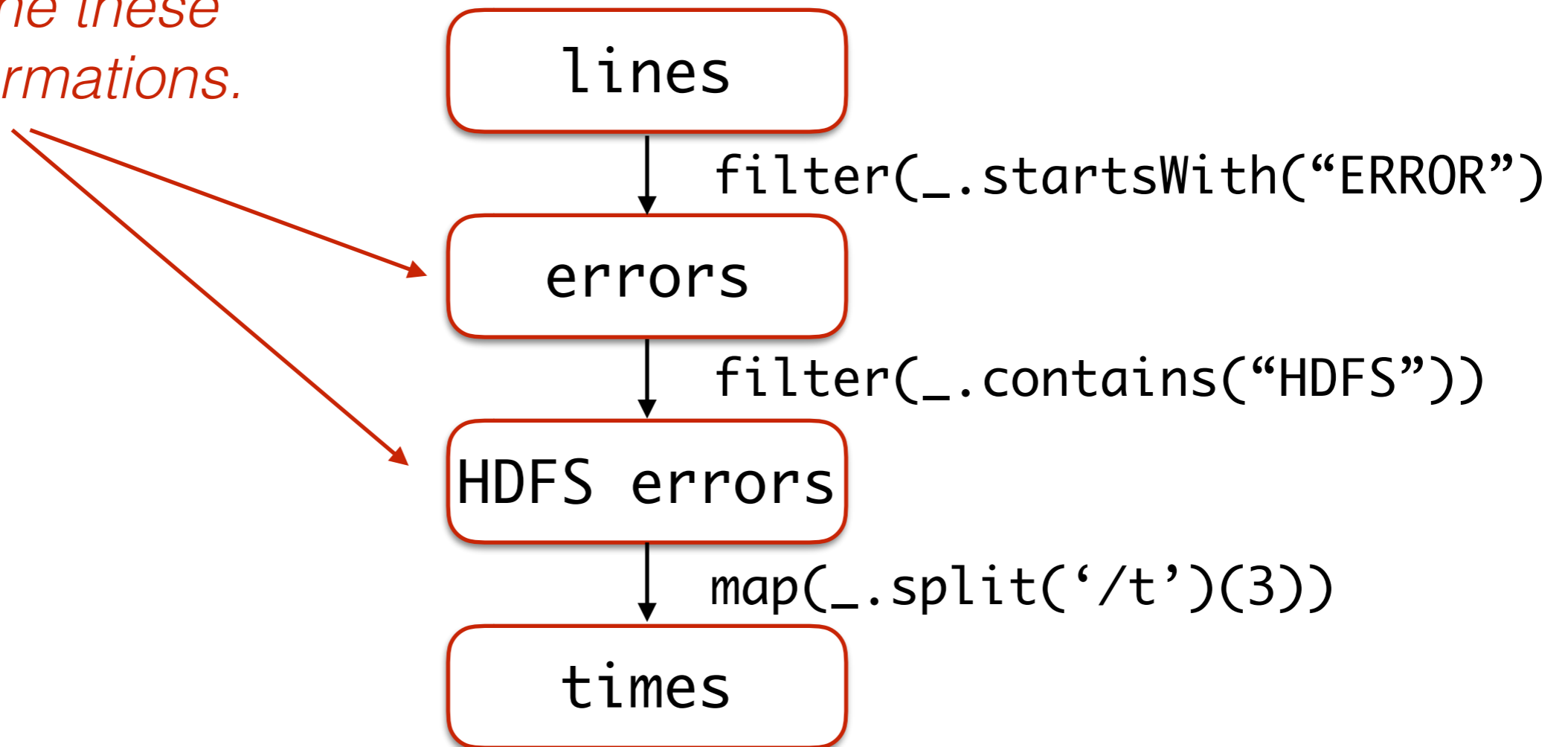
Alternative syntax using for- expressions.

Lineage Graph For times RDD



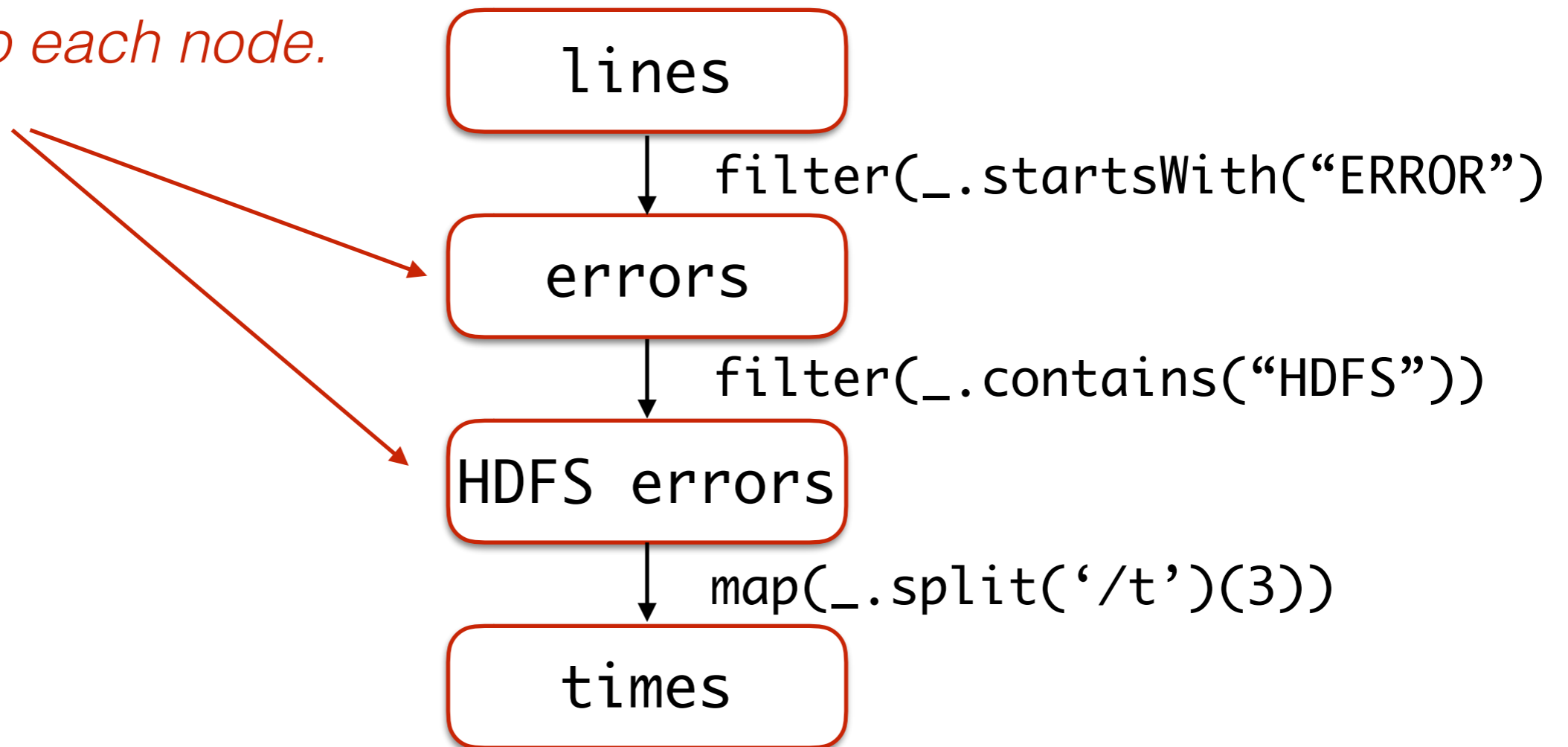
Lineage Graph For times RDD

*The Spark scheduler will
pipeline these
transformations.*



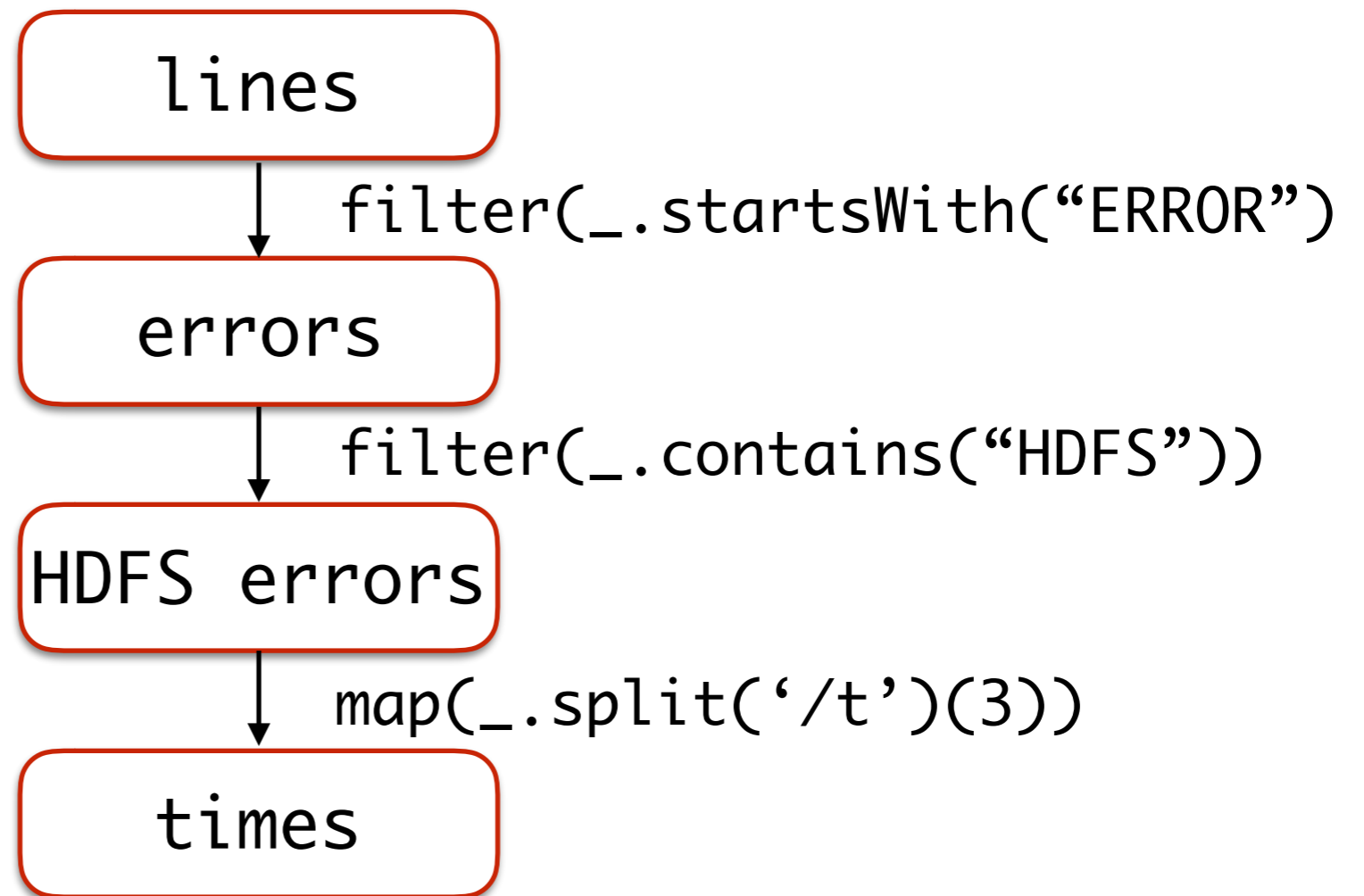
Lineage Graph For times RDD

*Tasks for transformations
are sent to each node.*



Lineage Graph For times RDD

*If a partition of errors
is lost, Spark rebuilds
it by recomputing from
the corresponding
partition of lines.*



Transformations Available on RDDs

- `map(f: T => U): RDD[T] => RDD[U]`
- `filter(f: T => Boolean): RDD[T] => RDD[T]`
- `flatMap(f: T => Sequence[U]): RDD[T] => RDD[U]`
- `sample(fraction: Float): RDD[T] => RDD[T]`
- `union(RDD[T], RDD[T]): RDD[T]`

Transformations Available On RDDs of Key/Value Pairs

`groupByKey(): RDD[(K, V)] => RDD[(K, Sequence[V])]`

`reduceByKey(f: (V, V) => V): RDD[(K, V)] => RDD[(K, V)]`

`join(): (RDD[(K, V)], RDD[(K, W)]) => RDD[K, (V, W)]`

Transformations Available On RDDs of Key/Value Pairs

`cogroup()`:

$(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow$

$\text{RDD}[K (\text{Sequence}[V], \text{Sequence}[W])]$

`crossProduct()`:

$(\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)]$

Transformations Available On RDDs of Key/Value Pairs

```
mapValues(f: V => W): RDD[(K,V)] => RDD[(K,W)]
```

Preserves partitioning.

Transformations Available On RDDs of Key/Value Pairs

```
partitionBy(p: Partitioner[K]): RDD[(K,V)] => RDD[(K,V)]
```

Actions Available on RDDs

```
count(): RDD[T] => Long  
collect(): RDD[T] => Sequence[T]  
reduce(f: (T,T) => T): RDD[T] => T  
lookup(k: K): RDD[(K,V)] => Sequence[V]  
save(path: String): ()
```


WordCount in Spark

```
val file = spark.textFile("hdfs://...")

val counts = file.flatMap(line => line.split(" "))
                  .map(word => (word, 1))
                  .reduceByKey((x,y) => x + y)

counts.saveAsTextFile("hdfs://...")
```

WordCount in Spark

```
val file = spark.textFile("hdfs://...")
```

```
val words = for (line <- file,  
                 word <- line.split(" "))  
              yield (word, 1)
```

```
val counts = words.reduceByKey(_ + _)
```

```
counts.saveAsTextFile("hdfs://...")
```

```
("this", "is", "a", "line", "this", "is",  
  "another", "line", "this", "is", "yet",  
  "another", "line")  
.map(word => (word, 1))
```

((“this”,1), (“is”,1), (“a”,1), (“line”,1),
 (“this”,1), (“is”,1), (“another”,1),
 (“line”,1), (“this”,1), (“is”,1),
 (“yet”,1), (“another”,1), (“line”,1))

```
x.reduceByKey(f) = x.groupByKey()  
                  .map(xs => xs.reduce(f))
```

```
((("this",1), ("is",1), ("a",1), ("line",1)),  
 ("this",1), ("is",1), ("another",1),  
 ("line",1), ("this",1), ("is",1),  
 ("yet",1), ("another",1), ("line",1))  
.groupByKey().map(xs => xs.reduce(_ + _)))
```

```
((("this", (1,1,1)),  
  ("is", (1,1,1)),  
  ("a", (1)),  
  ("line", (1,1,1)),  
  ("another", (1,1)),  
  ("yet", (1))).map(xs => xs.reduce(_ + _)))
```

```
((“this”, (1,1,1)).reduce(_ + _),  
 (“is”, (1,1,1)).reduce(_ + _),  
 (“a”, (1)).reduce(_ + _),  
 (“line”, (1,1,1)).reduce(_ + _),  
 (“another”, (1,1)).reduce(_ + _),  
 (“yet”, (1)).reduce(_ + _))
```


((“this”, 3), (“is”, 3), (“a”, 1),
 (“line”, 3), (“another”, 2), (“yet”, 1))

Machine Learning With Spark

- Given a collection of examples with various attributes and a label, we wish to predict the labels for new examples:

<height, weight, age, systolic bp, diastolic bp>: **medicine?**

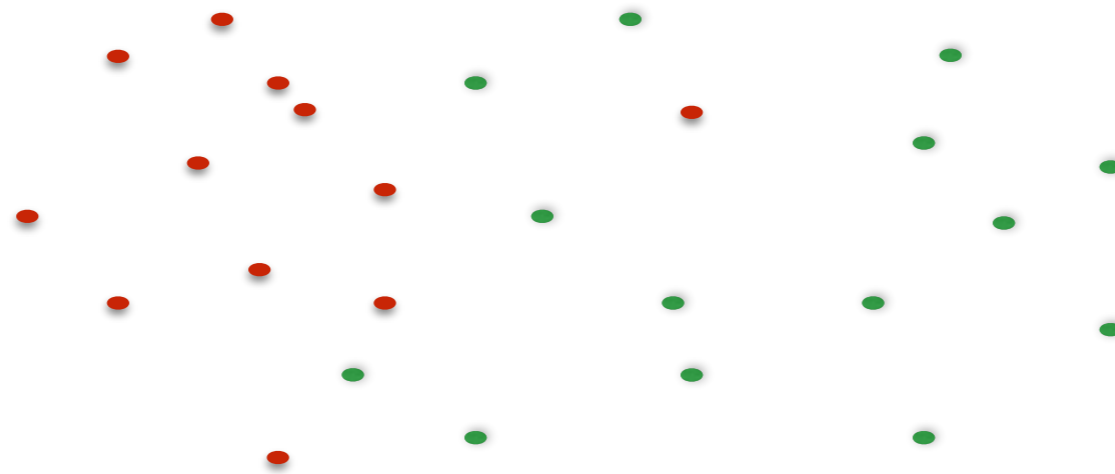
<170 cm, 72 kg, 52, 120, 80>: **YES**

<150 cm, 60 kg, 34 years, 130, 70> : **NO**

...

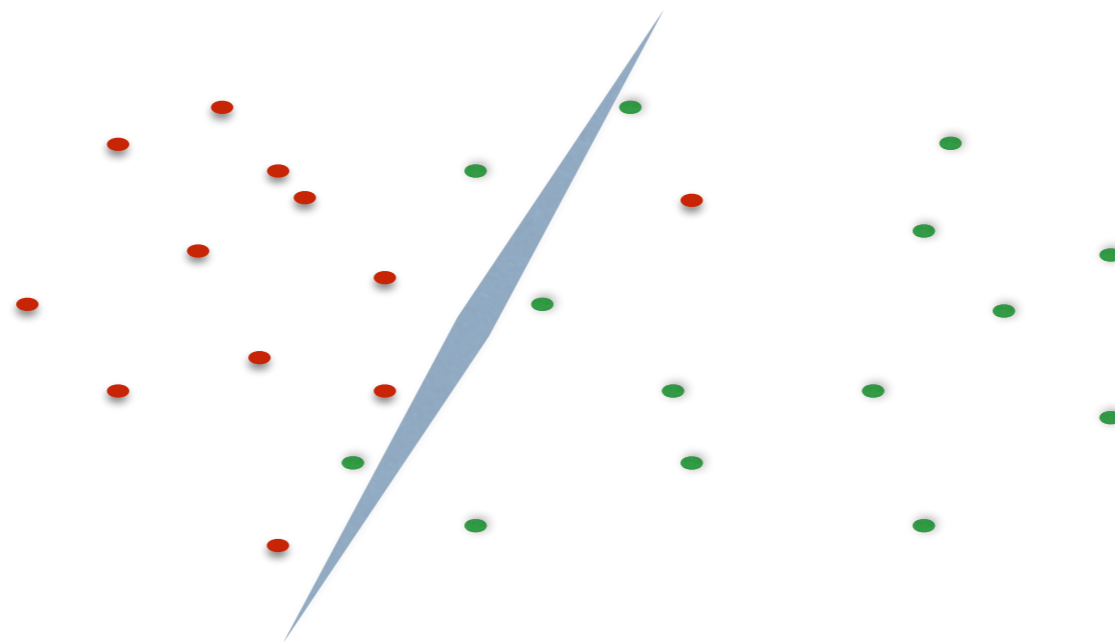
Machine Learning With Spark

- We can view the examples as vectors in a high-dimensional vector space
- The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels
- This new hyperplane can be used to predict labels for new examples



Machine Learning With Spark

- We can view the examples as vectors in a high-dimensional vector space
- The problem of labeling yes/no can be solved by finding the best hyperplane that divides the given examples according to their labels
- This new hyperplane can be used to predict labels for new examples



Logistic Regression With Spark

```
val points = spark.textFile(...).map(parsePoint).cache()

var w = Vector.random(D) // current separating plane

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)

  w -= gradient
}

println("Final separating plane: " + w)
```