CSCI-620

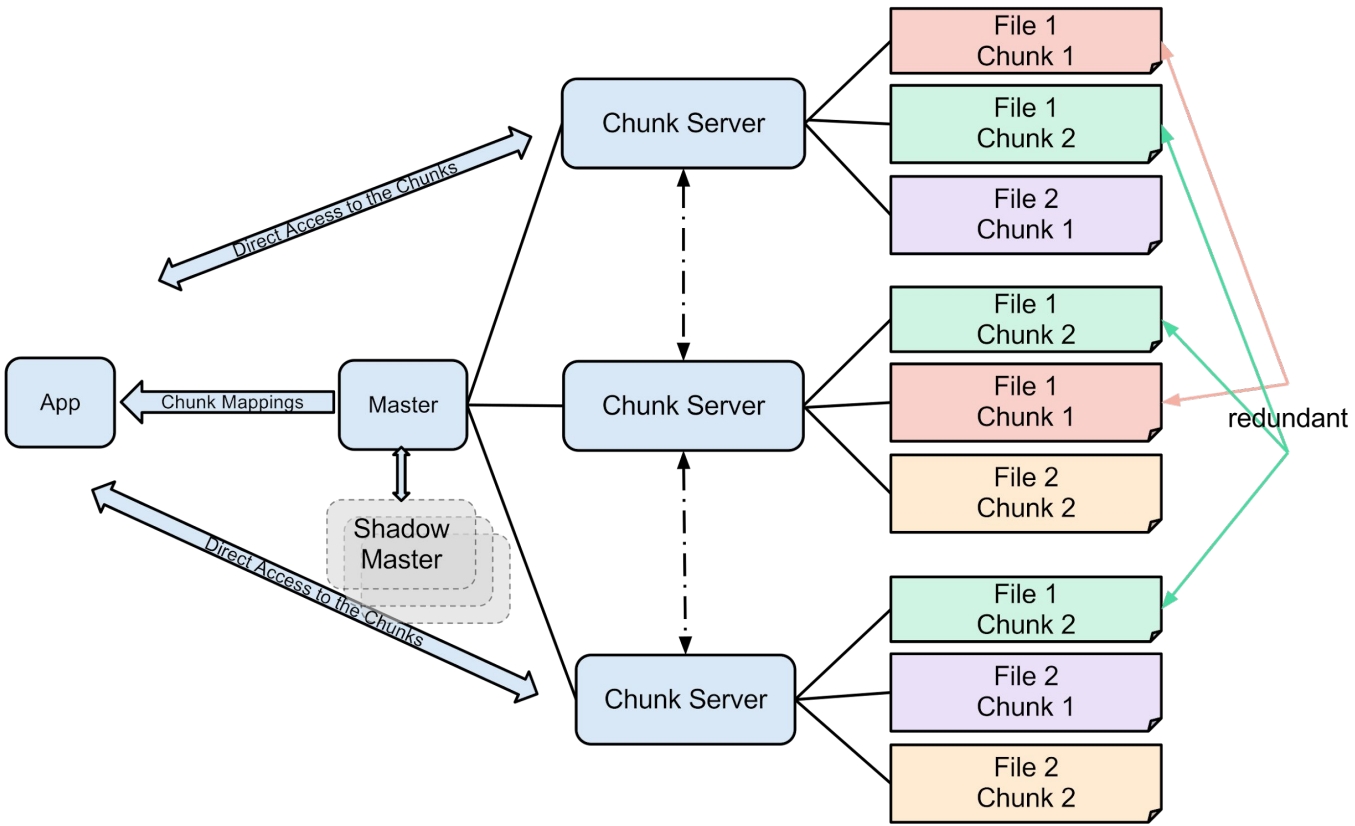
Distributed Data Processing

Distributed File Systems

- Many datasets are too large to fit on a single machine
- Unstructured data may not be easy to insert into a database
- Distributed file systems store data across a large number of servers

Google File System

- A distributed file system used by Google in the early 2000s
- Designed to run on a large number of cheap servers
- Somewhat slower, but tolerant of failures



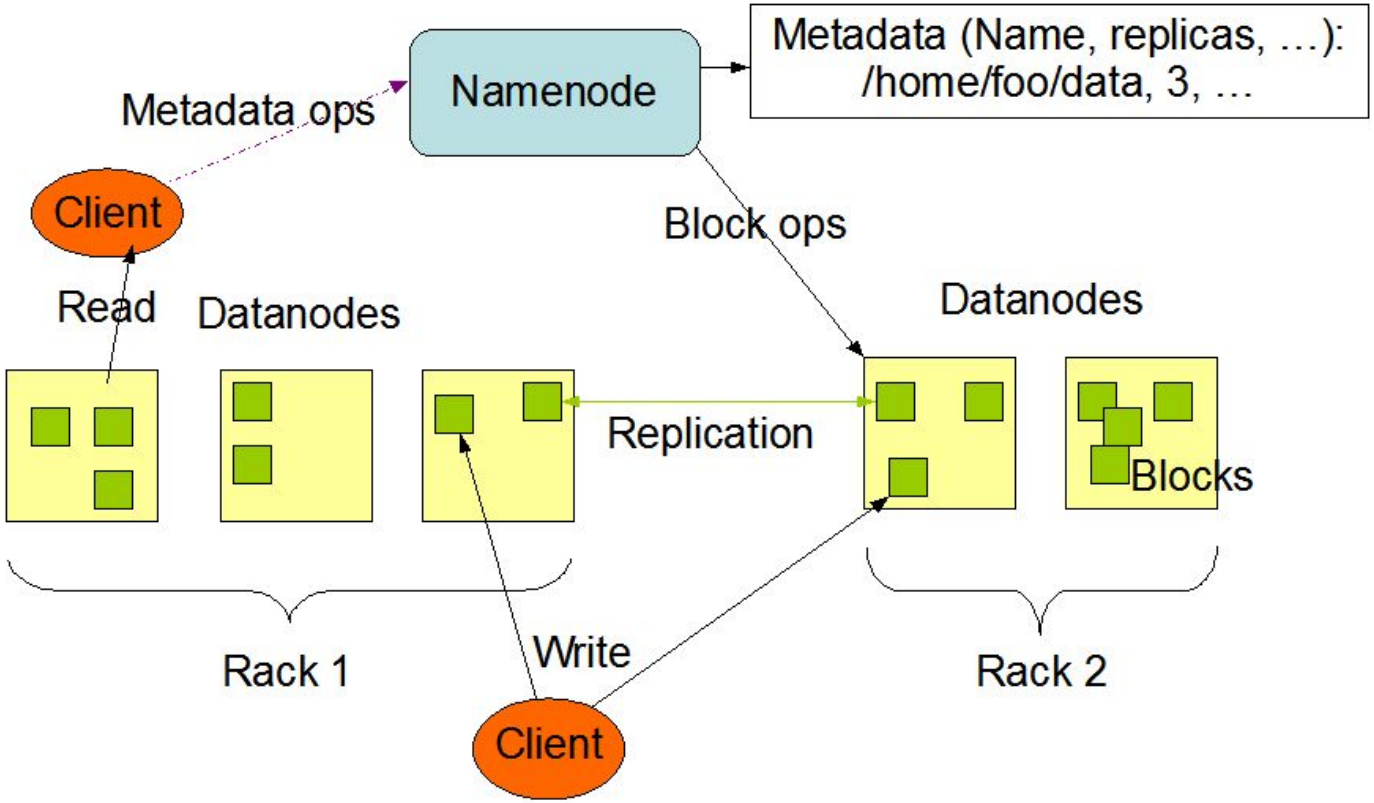
Google File System

History

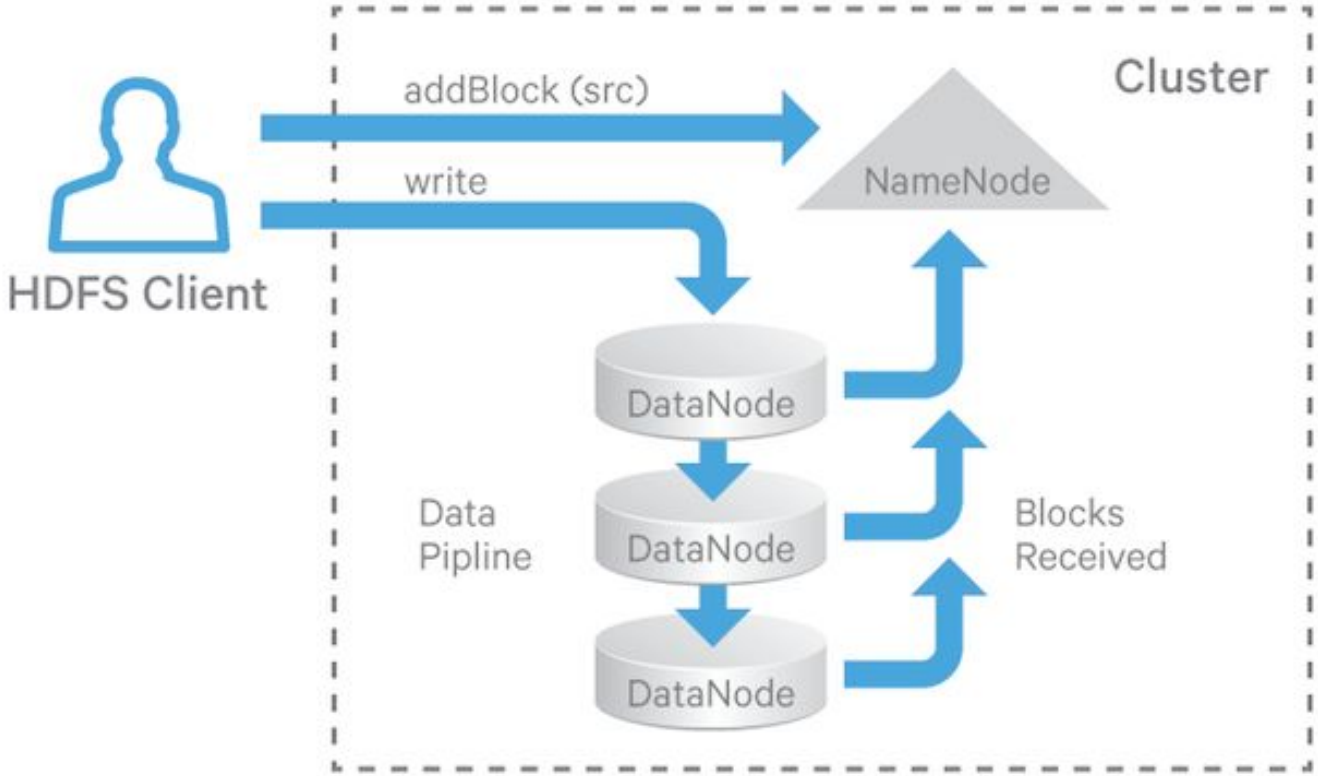
- 2003 Google File System
- 2004 MapReduce
- 2006 Hadoop
- 2007 HBase
- 2008 Hadoop wins TeraSort contest
- 2009 Spark
- 2009 Hadoop sorts one petabyte
- 2010 Hive

Hadoop File System

- Open source version of GFS
- Very similar architecture
- Deployments of 1,000s of nodes of HDFS exist



HDFS Architecture



HDFS Replication

Replication

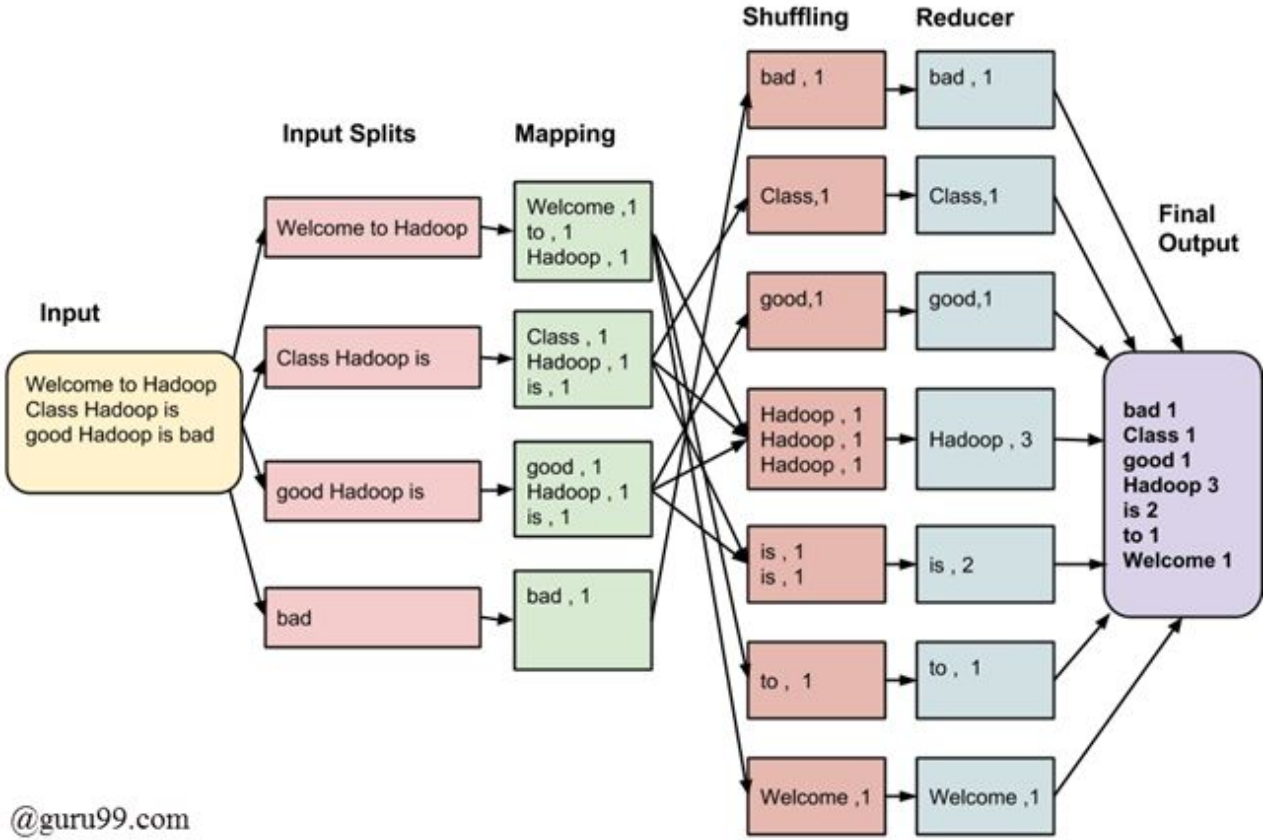
- Multiple copies are kept in case of failure so data isn't lost
- Can also makes reads faster since we have multiple copies to choose from
- Data is automatically split across multiple racks in a data center

MapReduce

- A programming model which consists of writing *map* and *reduce* functions
- Map accepts key/value pairs and produce a sequence of K/V pairs
- Data is shuffled to group keys together
- Reduce accepts values with the same key and produce a new K/V pair

MapReduce Execution

- Map tasks are assigned to machines based on the input data
- Map tasks produce their output
- Mapper output is shuffled and sorted
- Reduce tasks are scheduled and run
- Reduce output is stored to disk


Input
Welcome to Hadoop
Class Hadoop is
good Hadoop is bad
Input Splits
Welcome to Hadoop
Class Hadoop is
good Hadoop is
bad
Mapping
Welcome ,1
to , 1
Hadoop , 1
Class , 1
Hadoop , 1
is , 1
good , 1
Hadoop , 1
is , 1
bad , 1
Shuffling
bad , 1
Class,1
good,1
Hadoop , 1
Hadoop , 1
Hadoop , 1
is , 1
is , 1
to , 1
Welcome ,1
Reducer
bad , 1
Class,1
good,1
Hadoop , 3
is , 2
to , 1
Welcome ,1
Final
Output
bad 1
Class 1
good 1
Hadoop 3
is 2
to 1
Welcome 1
@guru99.com

MapReduce

```python
import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

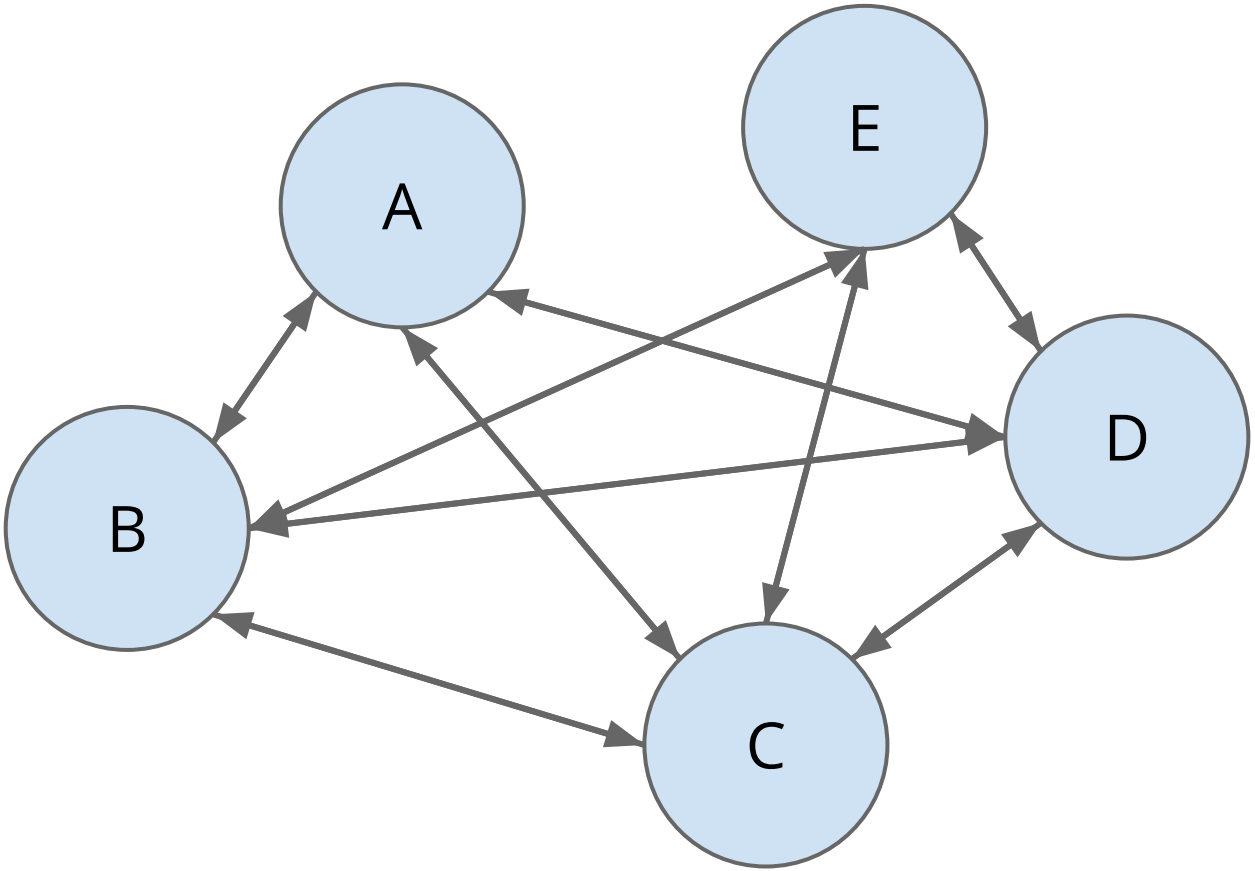
Map function

```python
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # (continued on next slide)
```

Reduce function (1/2)

```
# (continued within loop)

if current_word = word:
    current_count += int(count)
else:
    if current_word:
        print('%s\t%s' % (current_word, current_count)
    current_word = word
    current_count = count
```

Reduce function (2/2)



A → B C D
B → A C D E
C → A B D E
D → A B C E
E → B C D

Mutual friends

$A \rightarrow B\ C\ D$

A B	B C D
A C	B C D
A D	B C D

$B \rightarrow A\ C\ D\ E$

~~B A~~	~~A C D E~~
B C	A C D E
B D	A C D E
B E	A C D E

$C \rightarrow A\ B\ D\ E$

~~C A~~	~~A B D E~~
~~C B~~	~~A B D E~~
C D	A B D E
C E	A B D E

$D \rightarrow A\ B\ C\ E$

~~D A~~	~~A B C E~~
~~D B~~	~~A B C E~~
~~D C~~	~~A B C E~~
D E	A B C E

$E \rightarrow B\ C\ D$

~~E B~~	~~B C D~~
~~E C~~	~~B C D~~
~~E D~~	~~B C D~~

Map

$A \rightarrow B\ C\ D$

A B	B C D
A C	B C D
A D	B C D

$B \rightarrow A\ C\ D\ E$

A B	A C D E
B C	A C D E
B D	A C D E
B E	A C D E

$C \rightarrow A\ B\ D\ E$

A C	A B D E
B C	A B D E
C D	A B D E
C E	A B D E

$D \rightarrow A\ B\ C\ E$

A D	A B C E
B D	A B C E
C D	A B C E
D E	A B C E

$E \rightarrow B\ C\ D$

B E	B C D
C E	B C D
D E	B C D

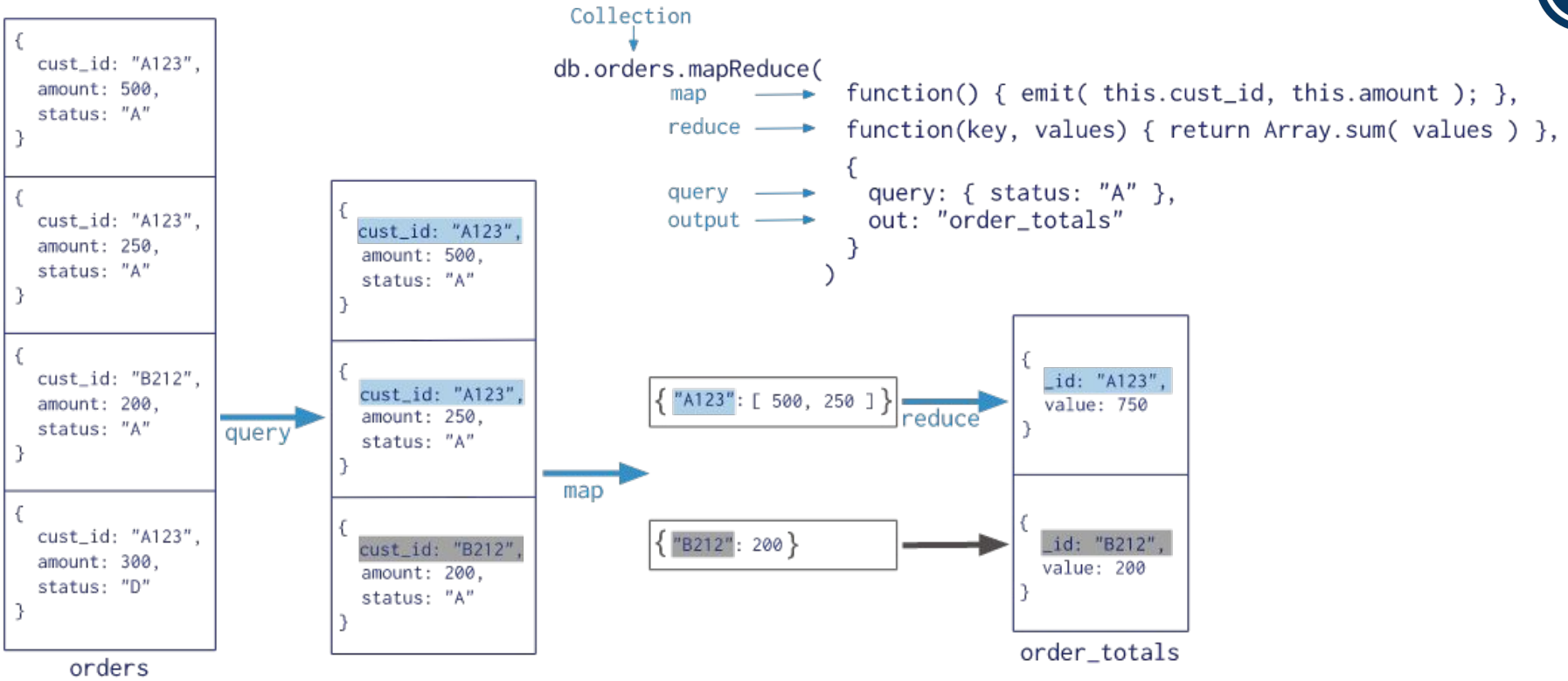
Map

A B	A C D E	B D	A B C E	D E	A B C E
A B	B C D	B D	A C D E	D E	B C D
A C	A B D E	B E	A C D E		
A C	B C D	B E	B C D		
A D	A B C E	C D	A B D E		
A D	B C D	C D	A B C E		
B C	A B D E	C E	A B D E		
B C	A C D E	C E	B C D		

Sorted Map Output

A B	C D	B D	A C E	D E	B C
A C	B D	B E	C D		
A D	B C	C D	A B E		
B C	A D E	C E	B D		

Reduce



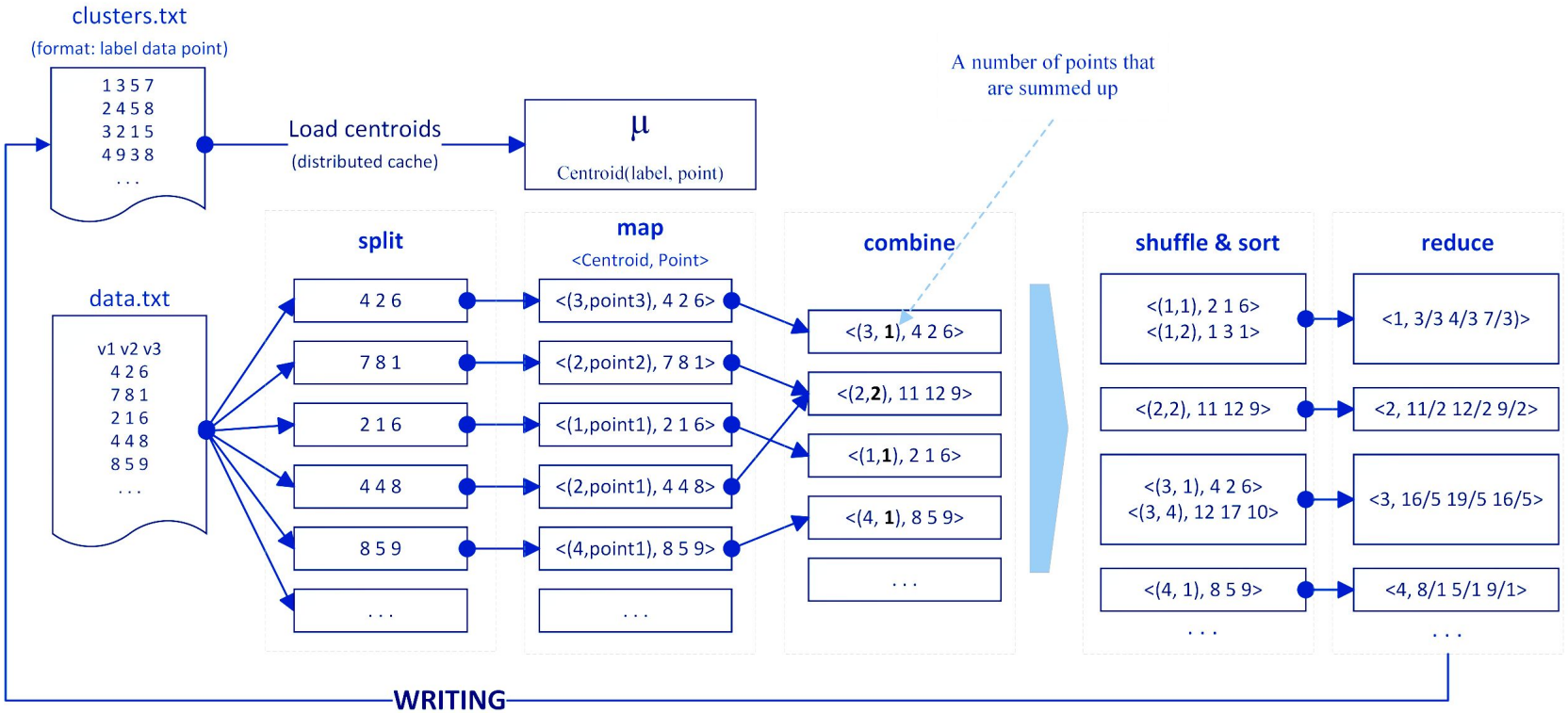
MapReduce in MongoDB

MapReduce vs Aggregation

- Aggregation pipelines are often easier to read than MapReduce programs
- MapReduce in MongoDB is generally less efficient and can't be optimized
- Collections in MongoDB can be "sharded" or split so MapReduce can happen in parallel as in Hadoop

Iterative MapReduce

- Sometimes an algorithm cannot be executed in a single MapReduce job
- MapReduce jobs can be chained together to provide a solution
- Often happens with algorithms which iterate until convergence
(e.g. k-means, PageRank, etc.)



MapReduce k-Means

Iterative MapReduce

- Reduce output must be read from disk again with each new job
- In some cases, the input data is read from disk many times
- There is no easy way to share the work done between iterations

Apache Spark

- Computation model is much richer than just MapReduce
- Transformations on input data can be written lazily and batched together
- Intermediate results can be cached and reused in future calculations

Spark Model

- Series of lazy **transformations** which are followed by **actions** that force evaluation of all transformations
- Each step produces a resilient distributed dataset (RDD)
- Intermediate results can be cached on memory or disk, optionally serialized

RDD Lineage

- For each RDD, we keep a *lineage*, the operations which created it
- Spark can then recompute any data which is lost without storing to disk
- We can still decide to keep a copy in memory or on disk for performance

Spark Model

Example

```
val in = spark.read.format("csv")
  .option("header", "true").csv("data.csv")
val over = in.filter($"y" > $"x").cache

val dist = over.select(hypot($"x", $"y"))
dist.write.csv("dist.csv")

val angle = over.select(atan($"y" / $"x"))
angle.write.csv("angle.csv")
```

Spark Model

Transformations

```
val in = spark.read.format("csv")
  .option("header", "true").csv("data.csv")
val over = in.filter($"y" > $"x").cache

val dist = over.select(hypot($"x", $"y"))
dist.write.csv("dist.csv")

val angle = over.select(atan($"y" / $"x"))
angle.write.csv("angle.csv")
```

Spark Model

Actions

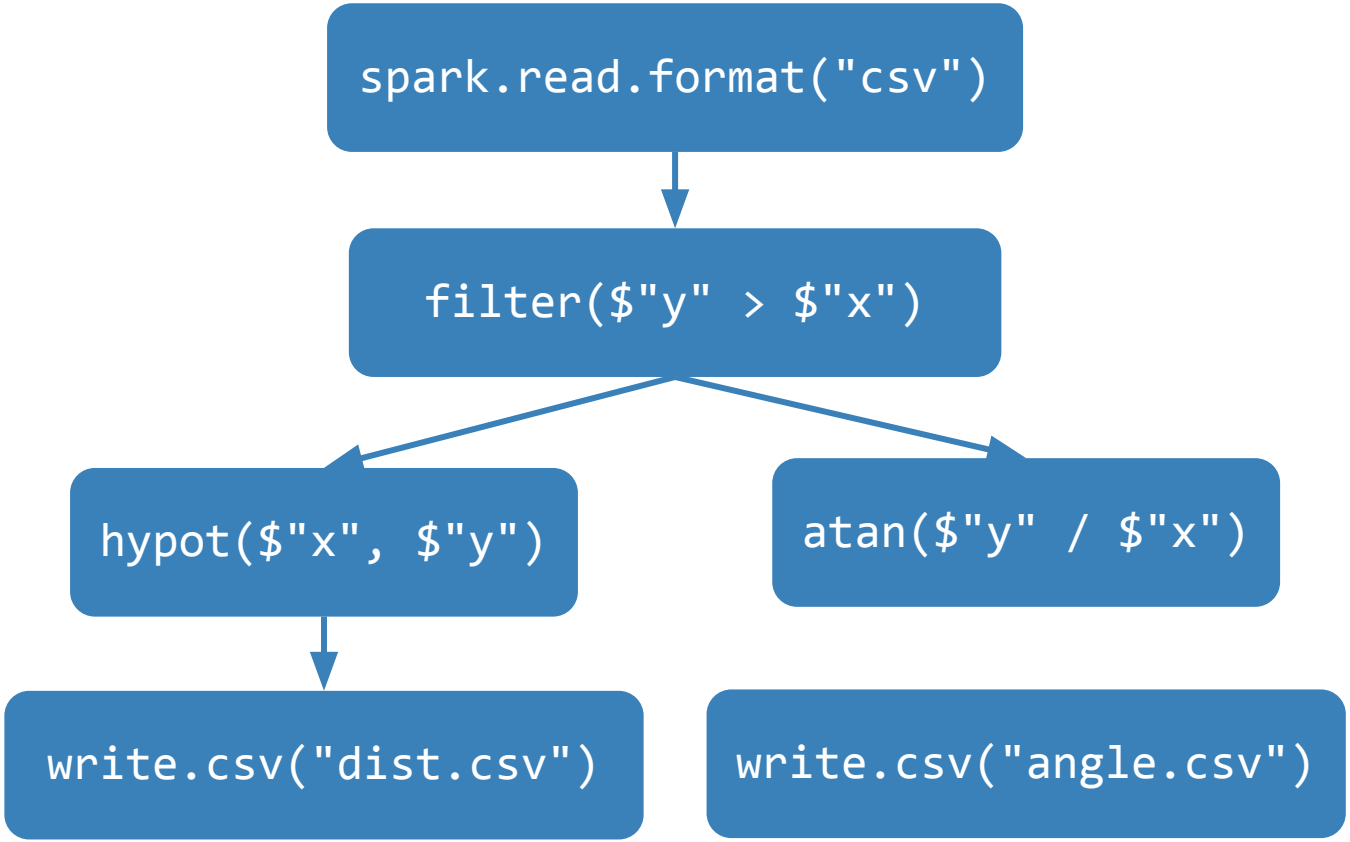
```
val in = spark.read.format("csv")
  .option("header", "true").csv("data.csv")
val over = in.filter($"y" > $"x").cache

val dist = over.select(hypot($"x", $"y"))
dist.write.csv("dist.csv")

val angle = over.select(atan($"y" / $"x"))
angle.write.csv("angle.csv")
```

Spark Transforms

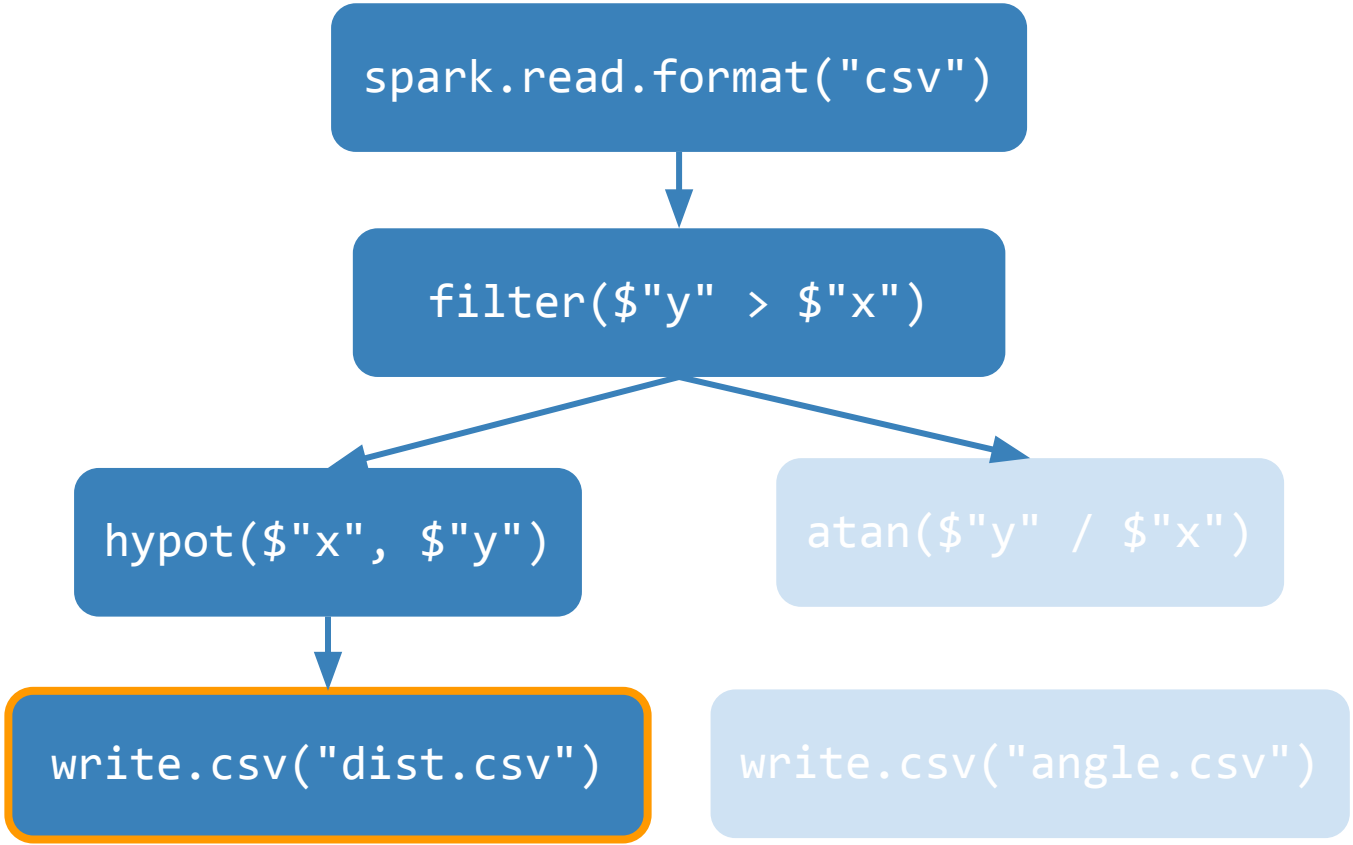
- distinct()
- filter(fn)
- intersection(other)
- join(other)
- map(fn)
- union(other)

Spark Actions

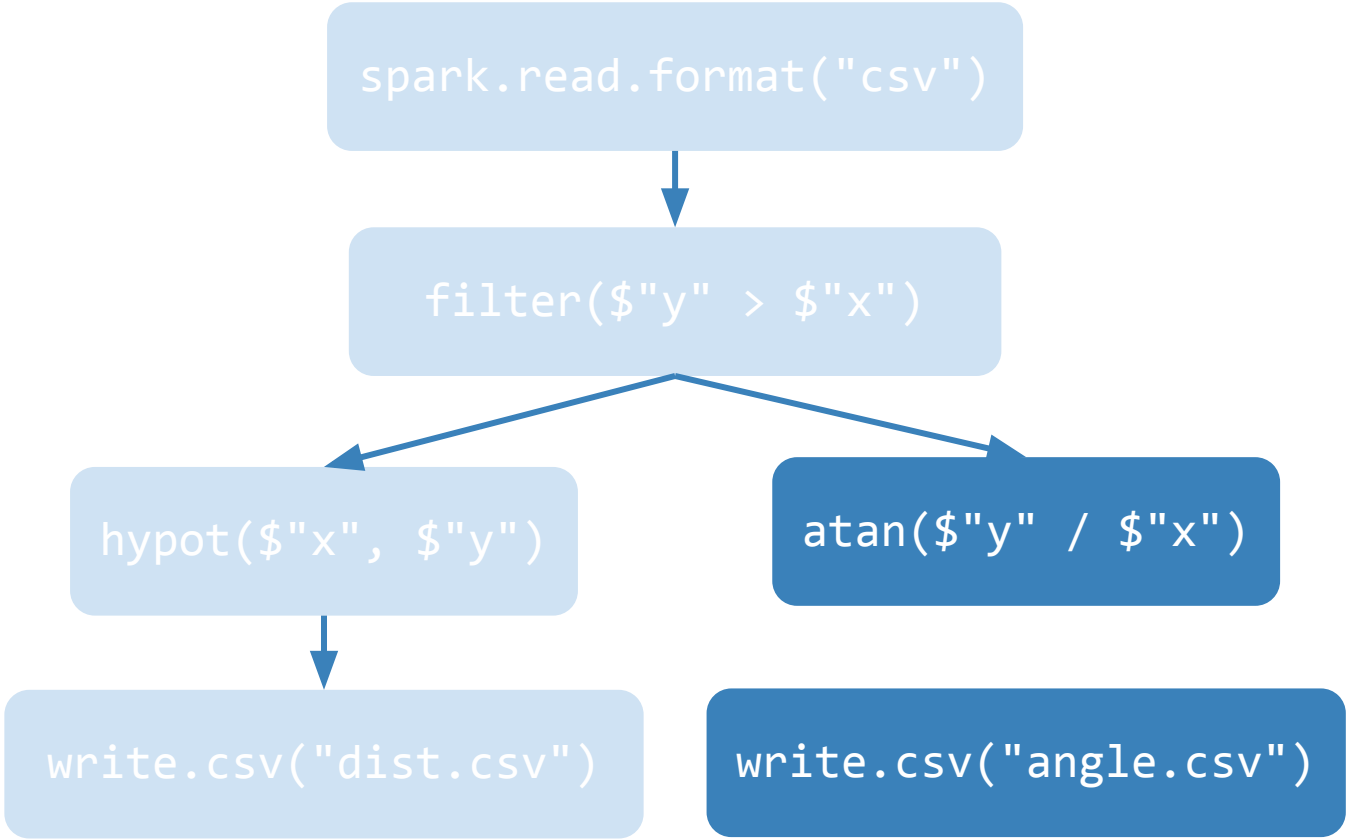
- collect()
- count()
- first()
- take(n)
- reduce(fn)
- foreach(fn)
- saveAsTextFile()


spark.read.format("csv")
filter($"y" > $"x")
hypot($"x", $"y")
atan($"y" / $"x")
write.csv("dist.csv")
write.csv("angle.csv")

Task DAG


spark.read.format("csv")
filter($"y" > $"x")
hypot($"x", $"y")
atan($"y" / $"x")
write.csv("dist.csv")
write.csv("angle.csv")

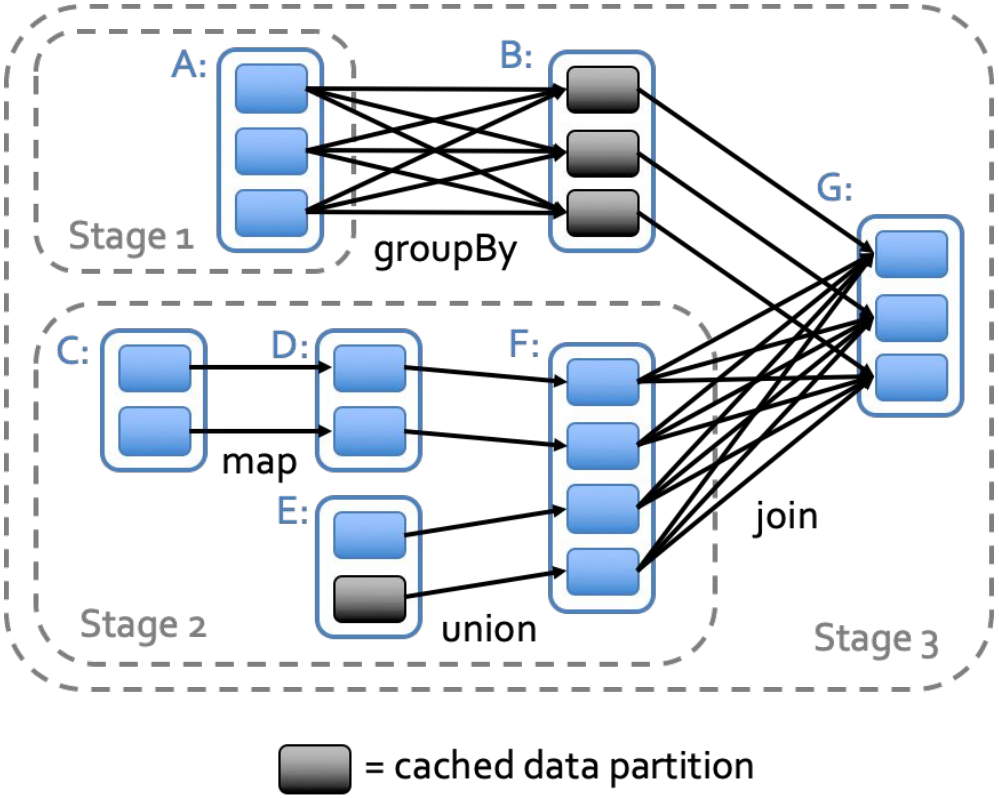
Task DAG


spark.read.format("csv")
filter($"y" > $"x")
hypot($"x", $"y")
atan($"y" / $"x")
write.csv("dist.csv")
write.csv("angle.csv")

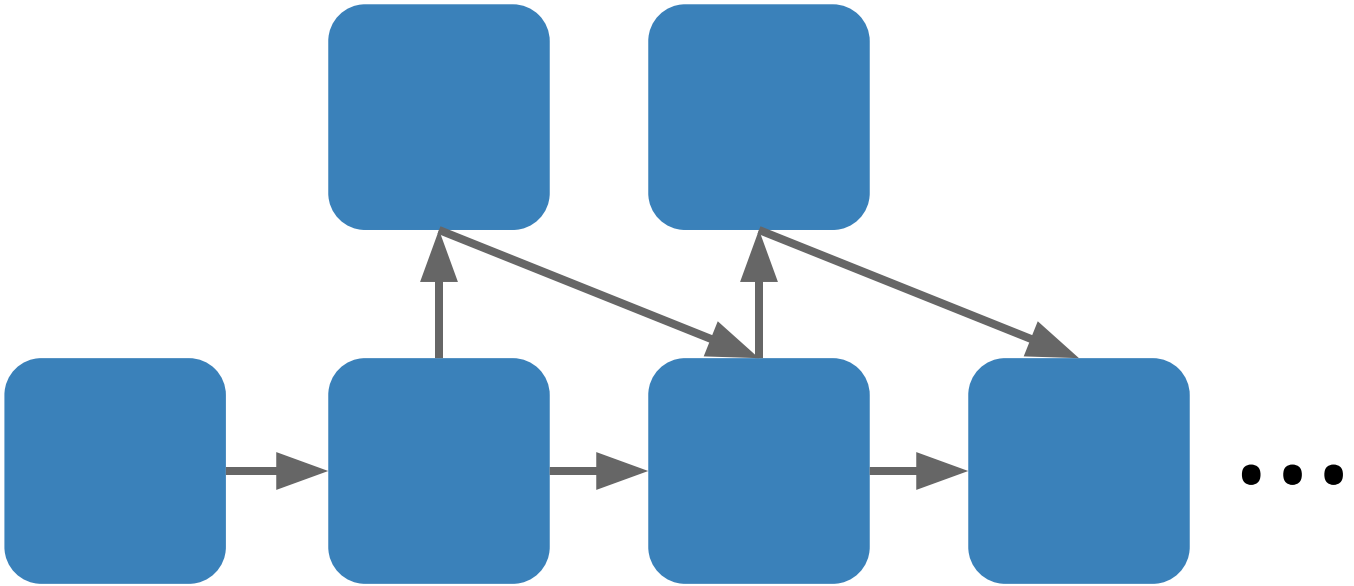
Task DAG

Spark Stages

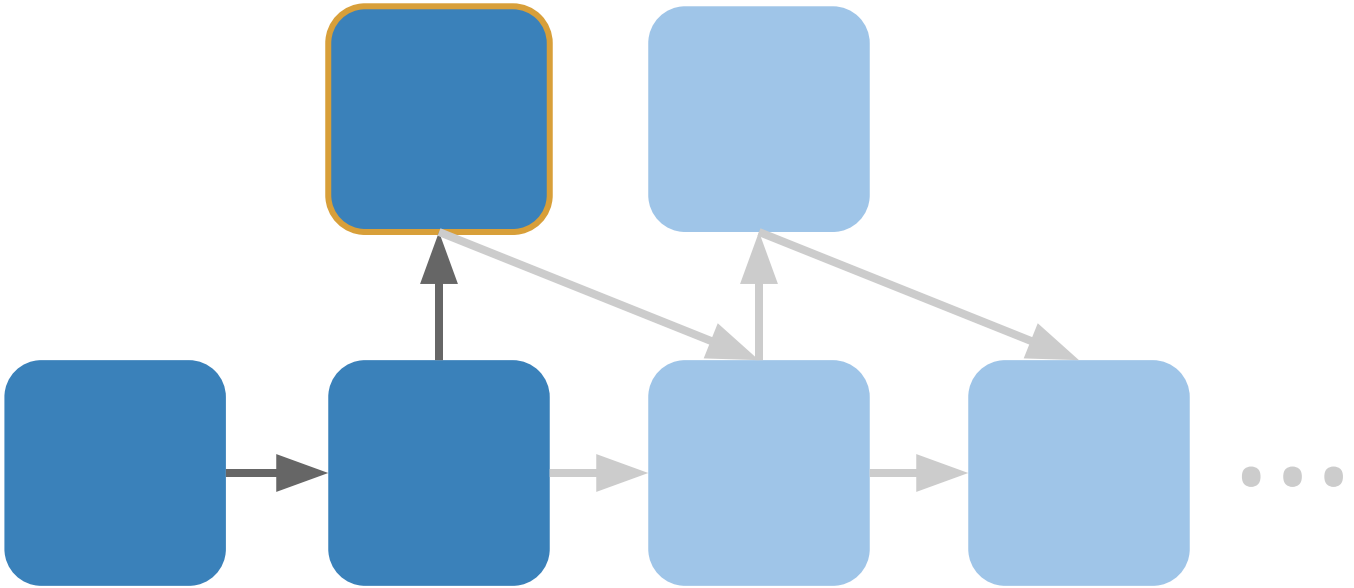
- Some operations such as join, sort and grouping, require shuffling
- This is similar to the sorting before the reduce phase in MapReduce
- Execution is grouped into *stages* depending on where shuffles occur
- Tasks execute on partitions in parallel



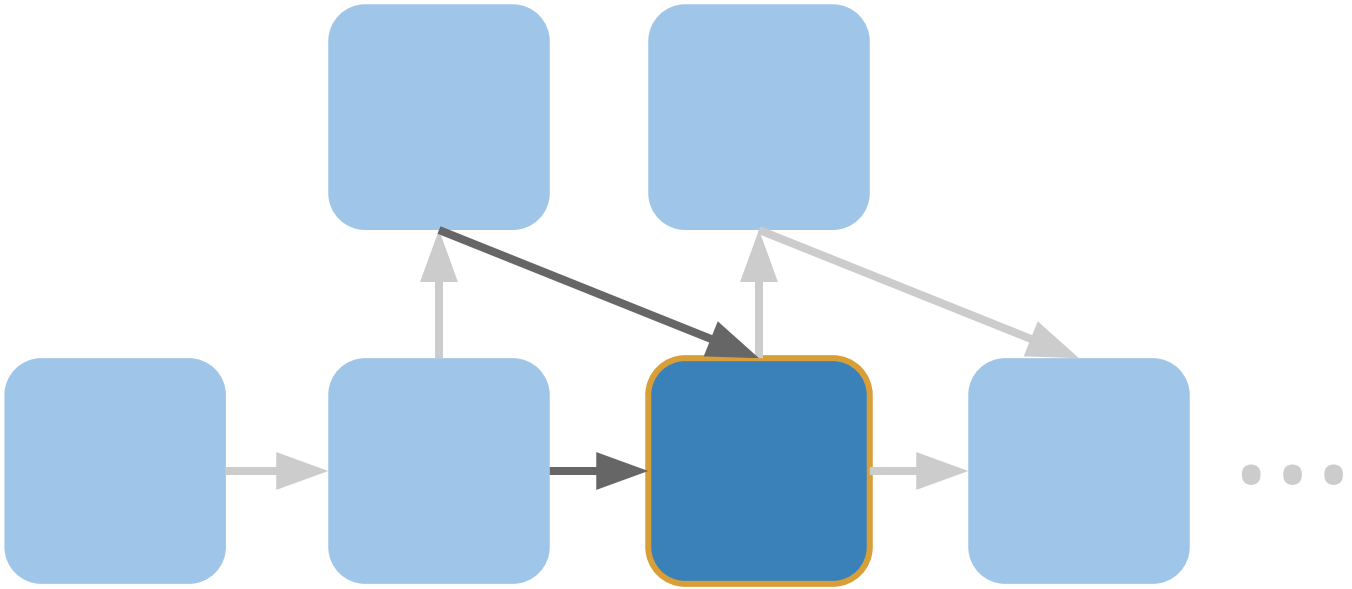
Spark Scheduler



Spark Iteration



Spark Iteration



Spark Iteration

Spark k-means

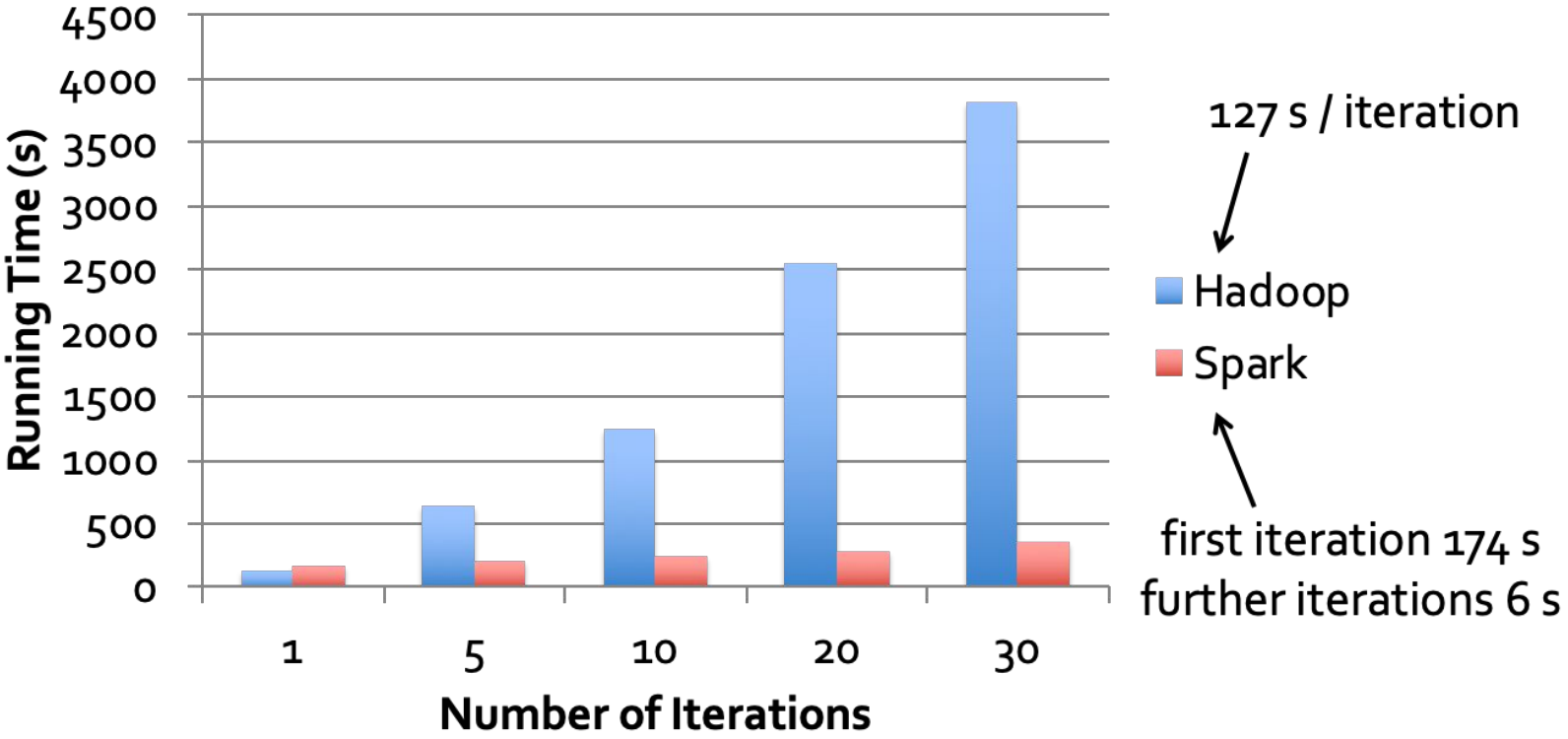
1. Load data and “persist”
2. Randomly sample for initial clusters
3. Broadcast the initial clusters
4. Loop over data and find new centers
5. Repeat from 3 until converged

Spark k-means

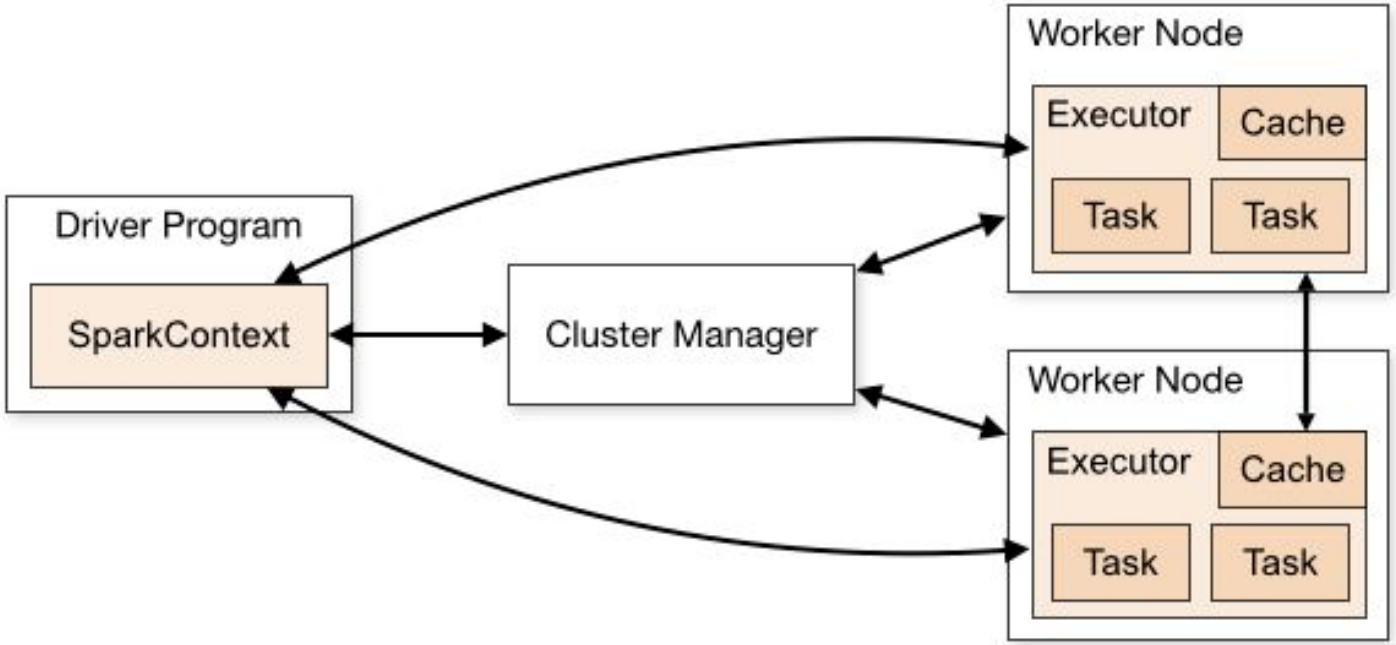
- Unlike MapReduce, we can keep the input in memory and load them once
- Broadcasting means we can quickly send the centers to all machines
- All cluster assignments do not need to be written to disk every time

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

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

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

Logistic Regression



Logistic Regression



Spark Architecture

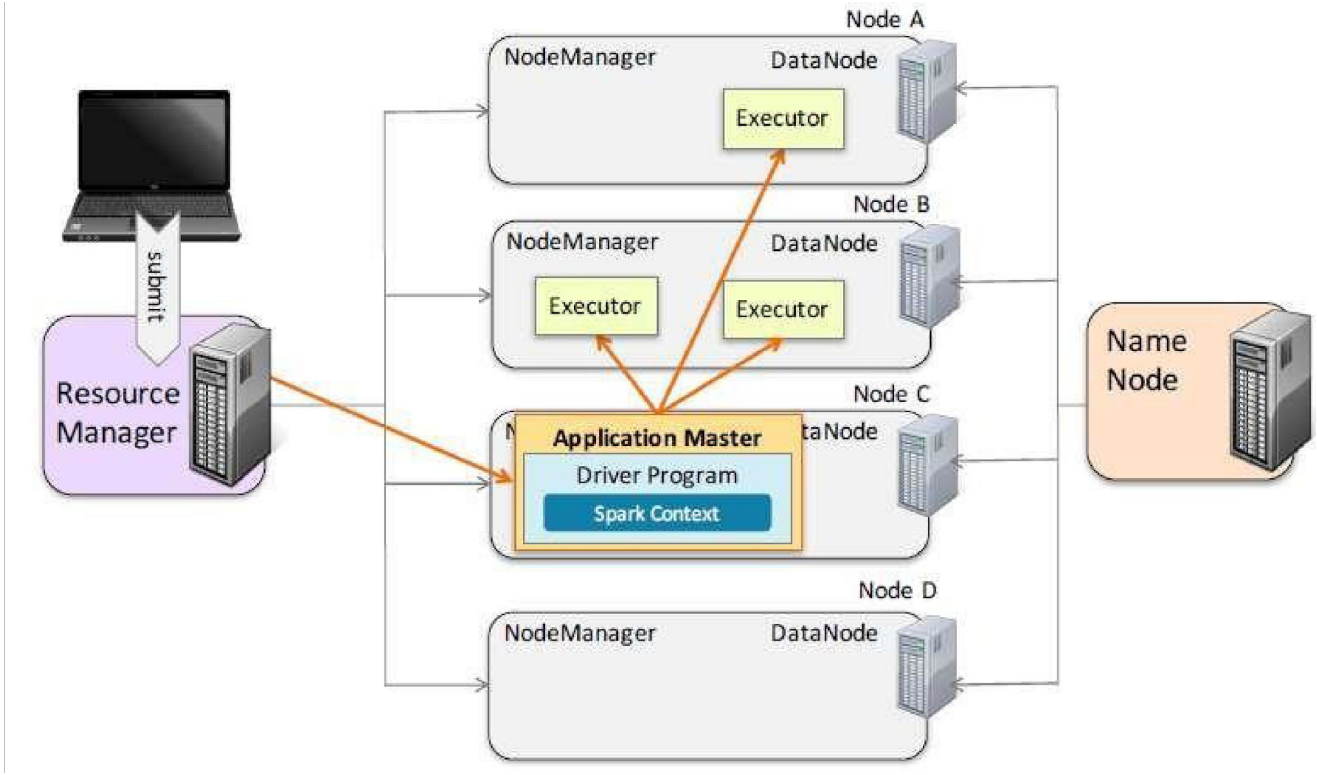
Spark Driver

- Runs the application to define RDDs and submit jobs to run on the cluster
- Creates a Spark *context* which represents the execution environment
- Tracks the progress of all jobs and displays a web interface

Spark Cluster Manager

- Multiple cluster managers can be used
  - Standalone - single machine
  - Apache Mesos
  - Hadoop YARN
  - Kubernetes
  - (and others)

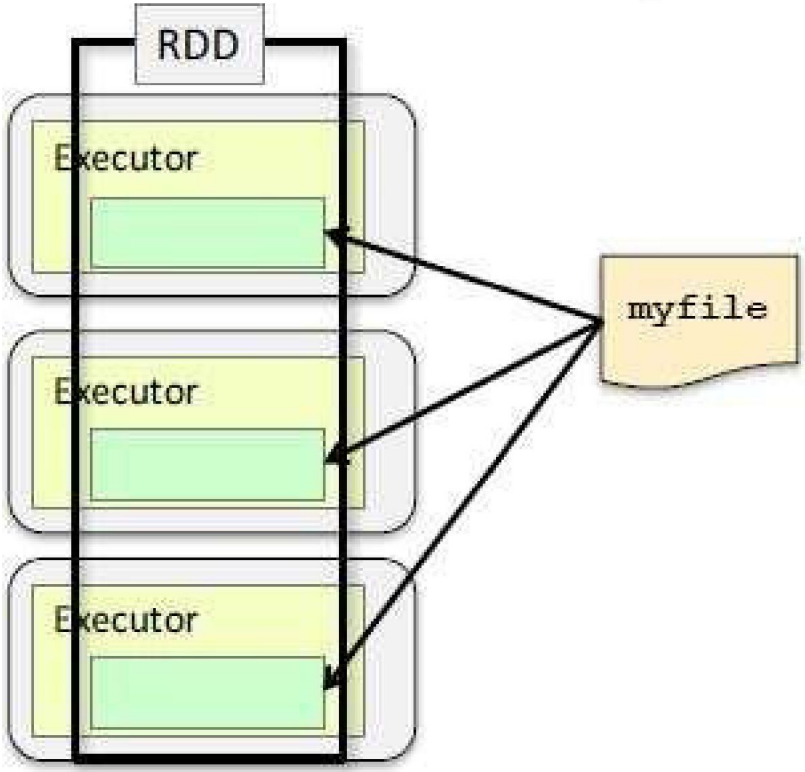
- Handles creating workers to run each distributed Spark job



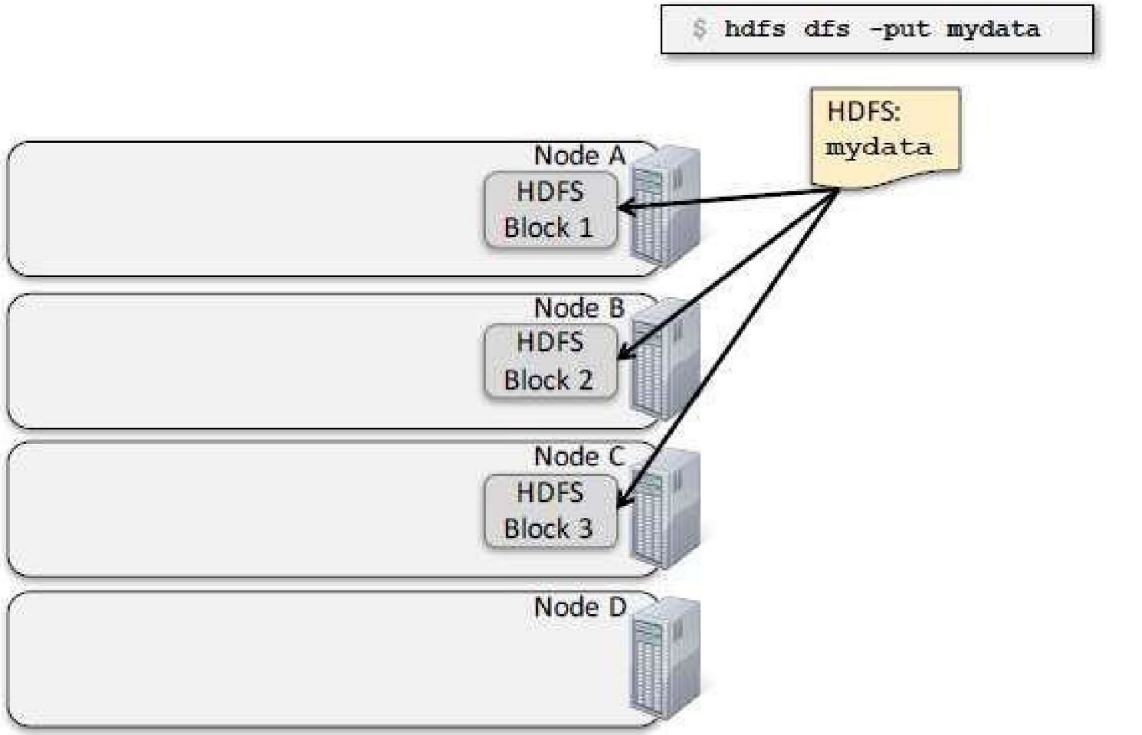
Spark on YARN

Spark Executors

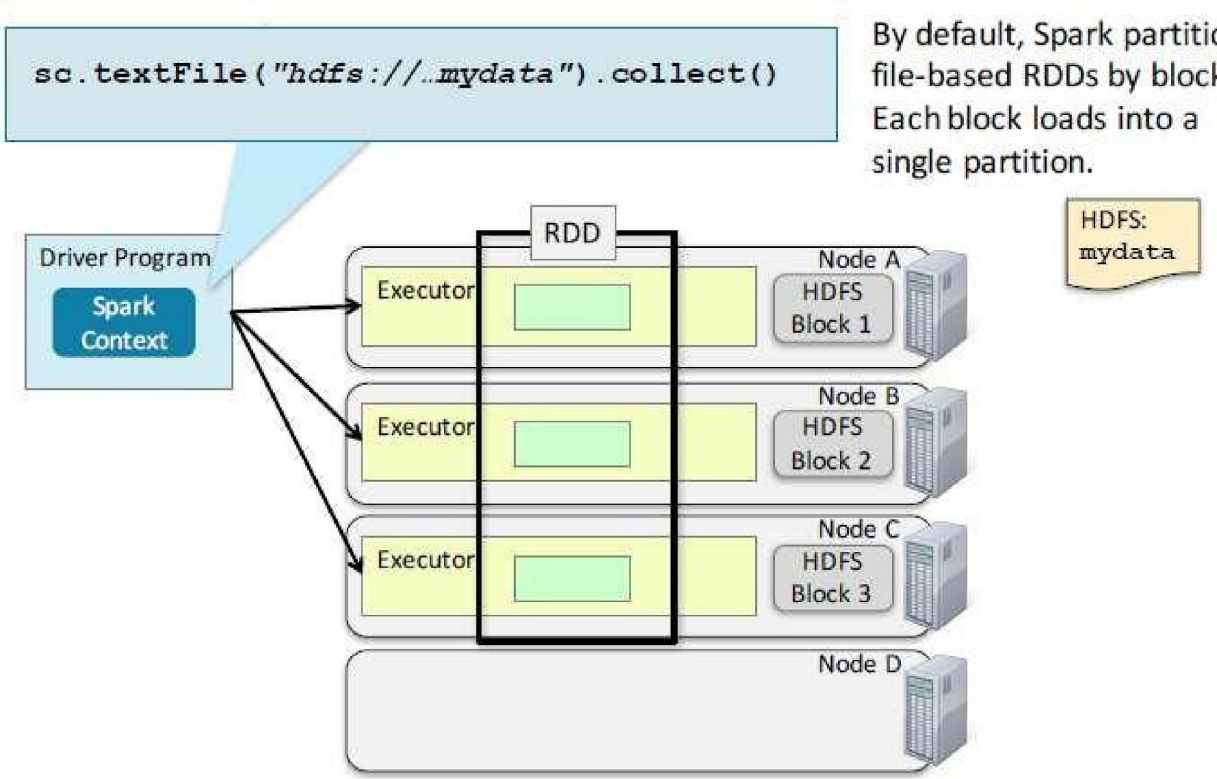
- Executors run across multiple machines in a Spark cluster
- Receives a series of transformations to execute on RDDs from the drivers
- If an executor fails, the driver can ask the cluster manager to start another



Parallelism in Spark



Parallelism in Spark

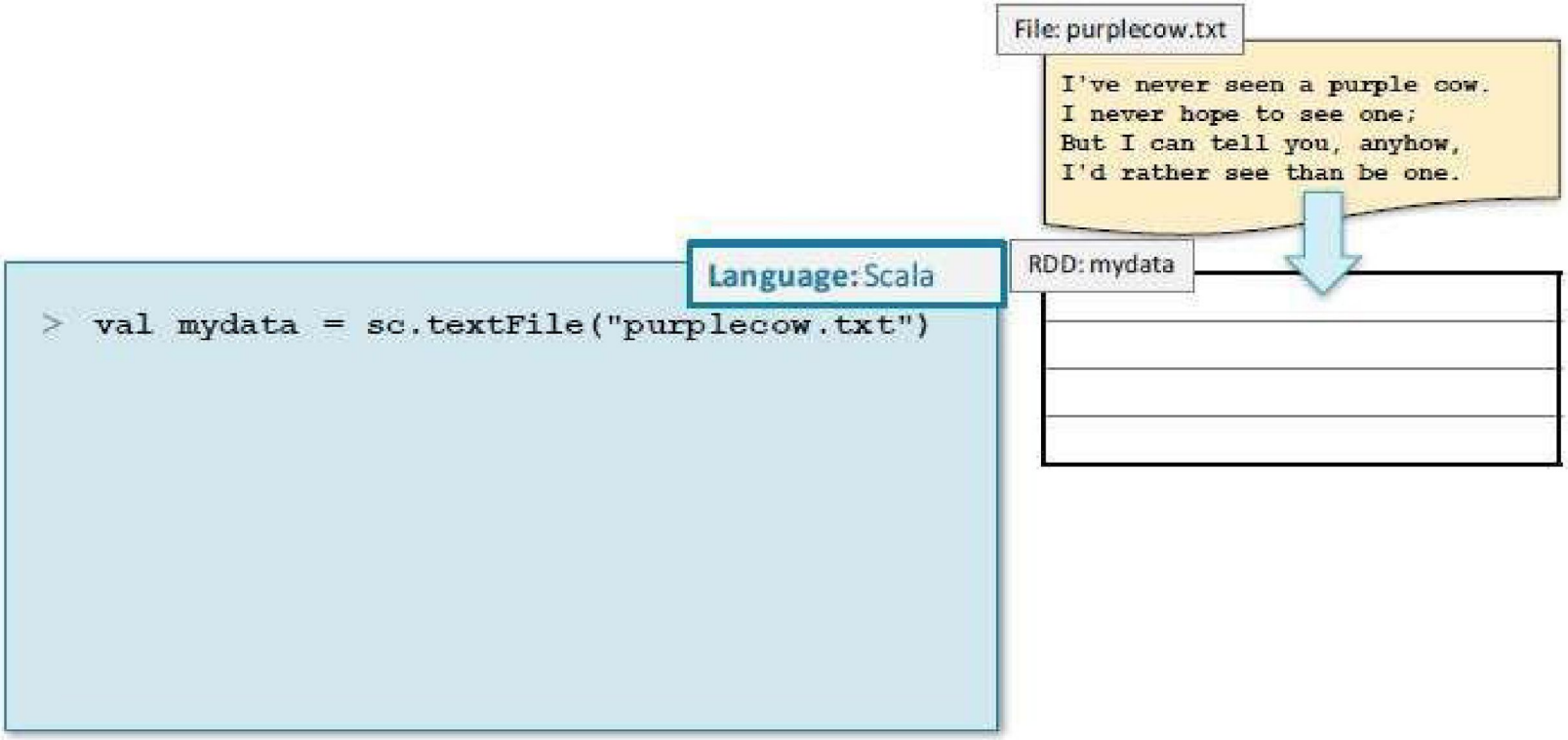


Parallelism in Spark

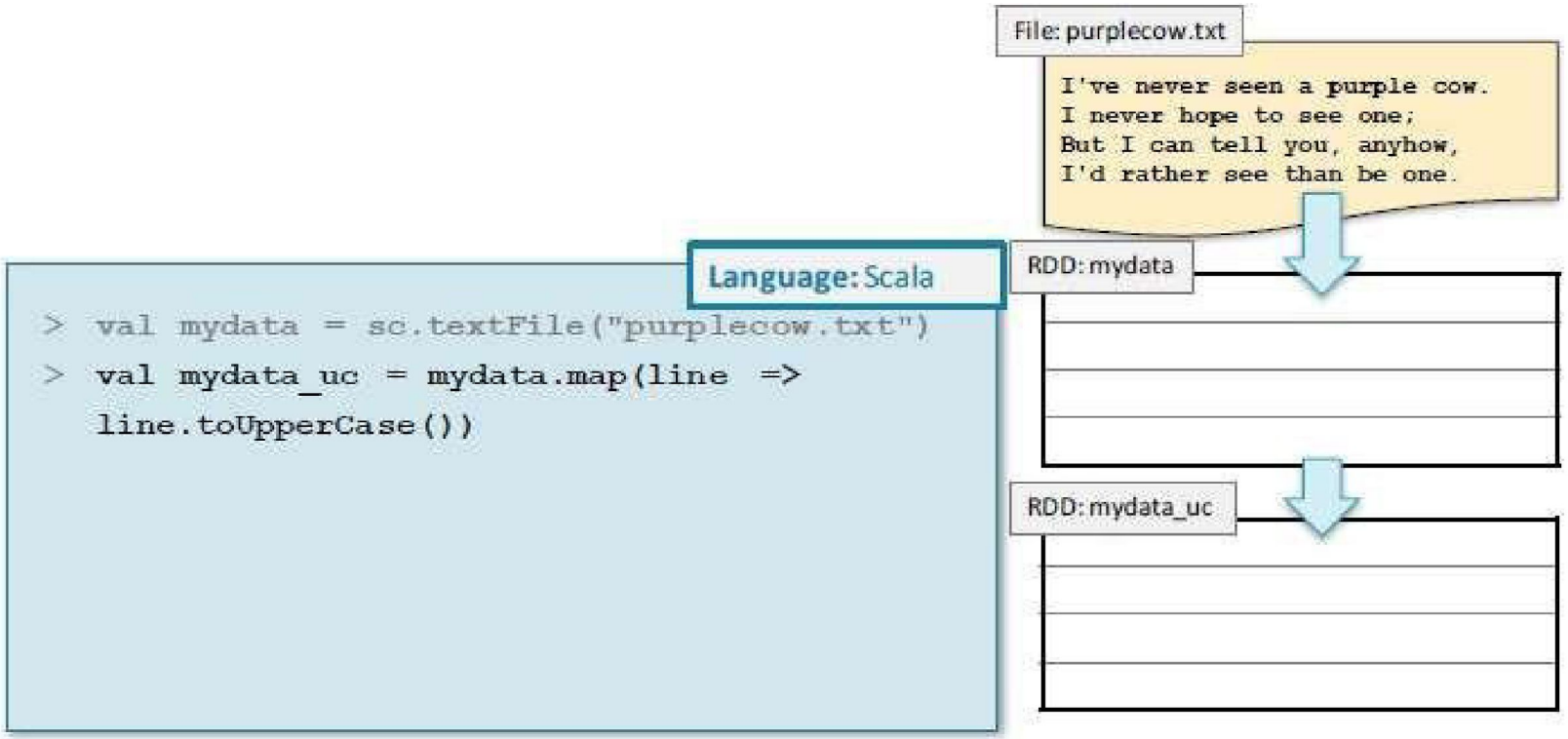
File: purplecow.txt

```
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.
```

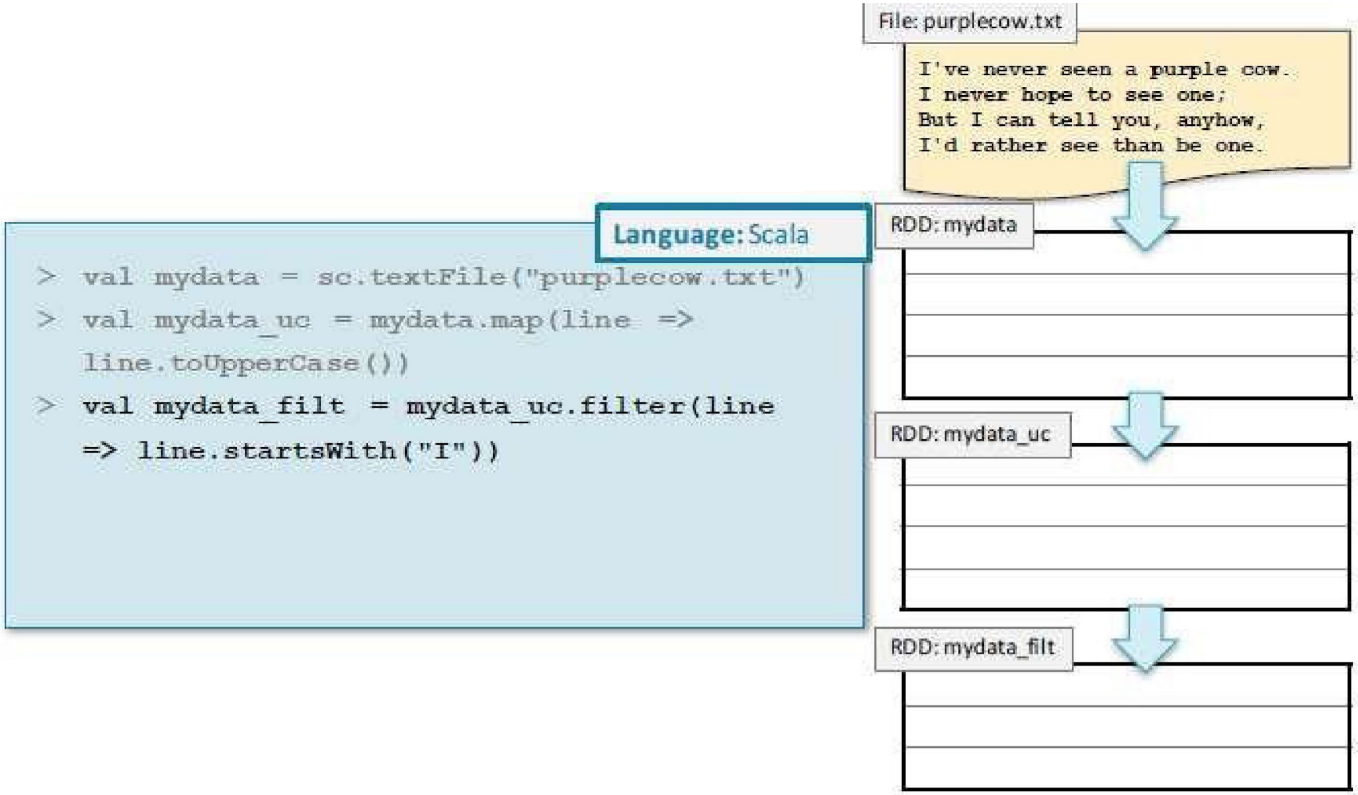
Spark Execution



Spark Execution



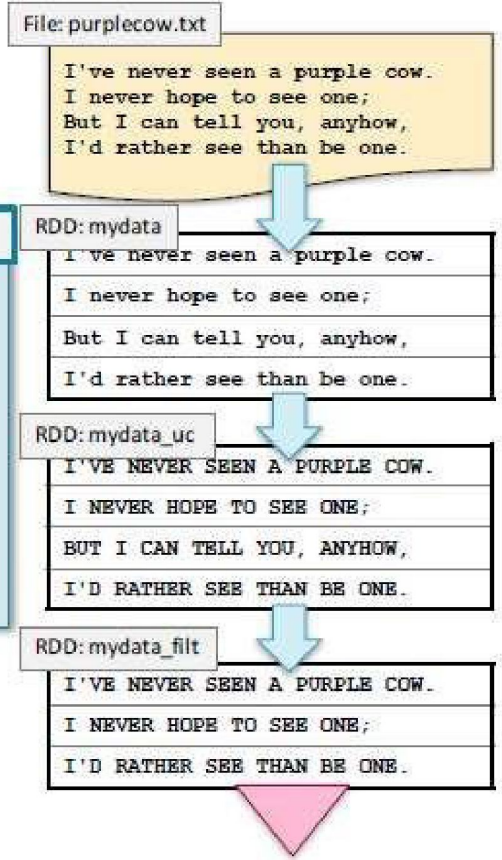
Spark Execution



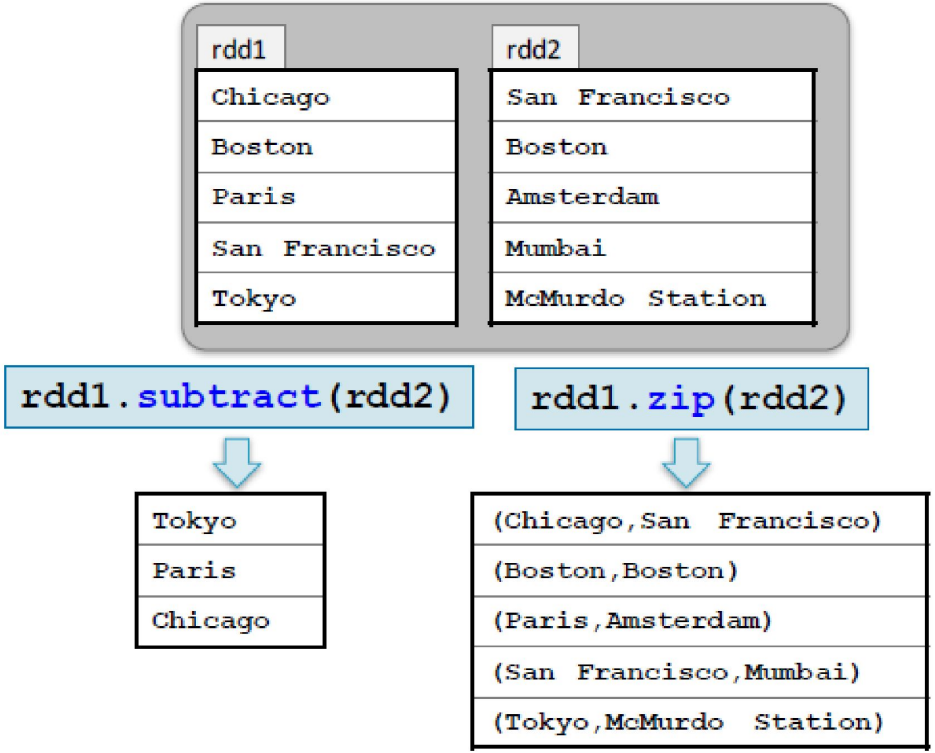
Spark Execution

Language: Scala

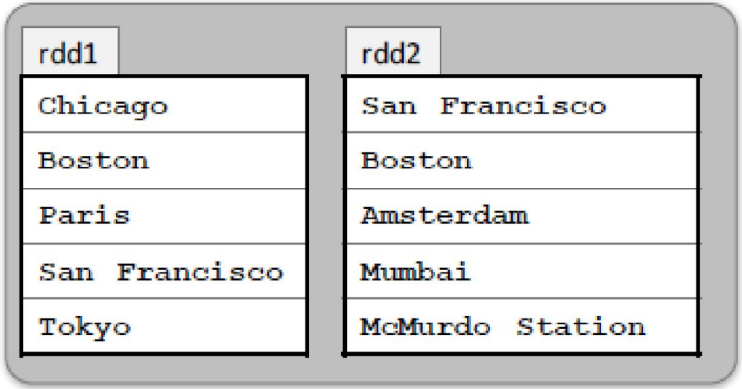
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Spark Execution



Multi-RDD Operations



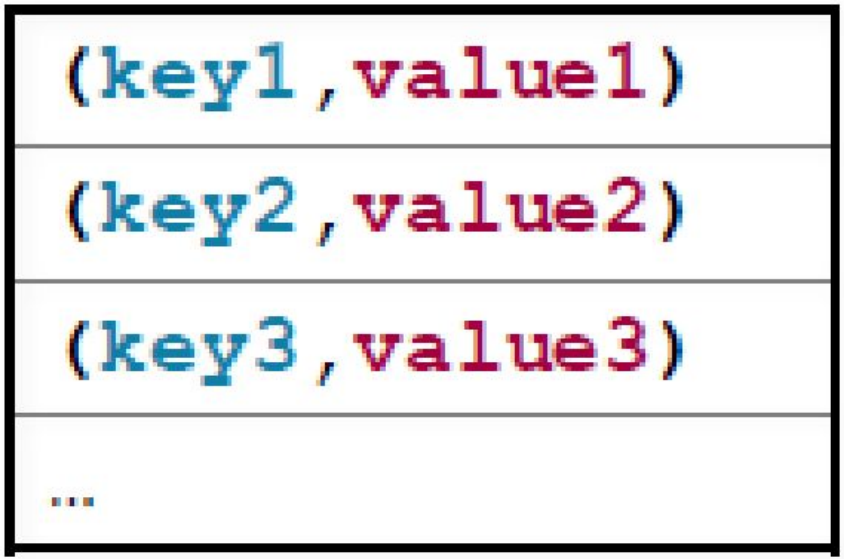
`rdd1.intersection(rdd2)`

Boston
San Francisco

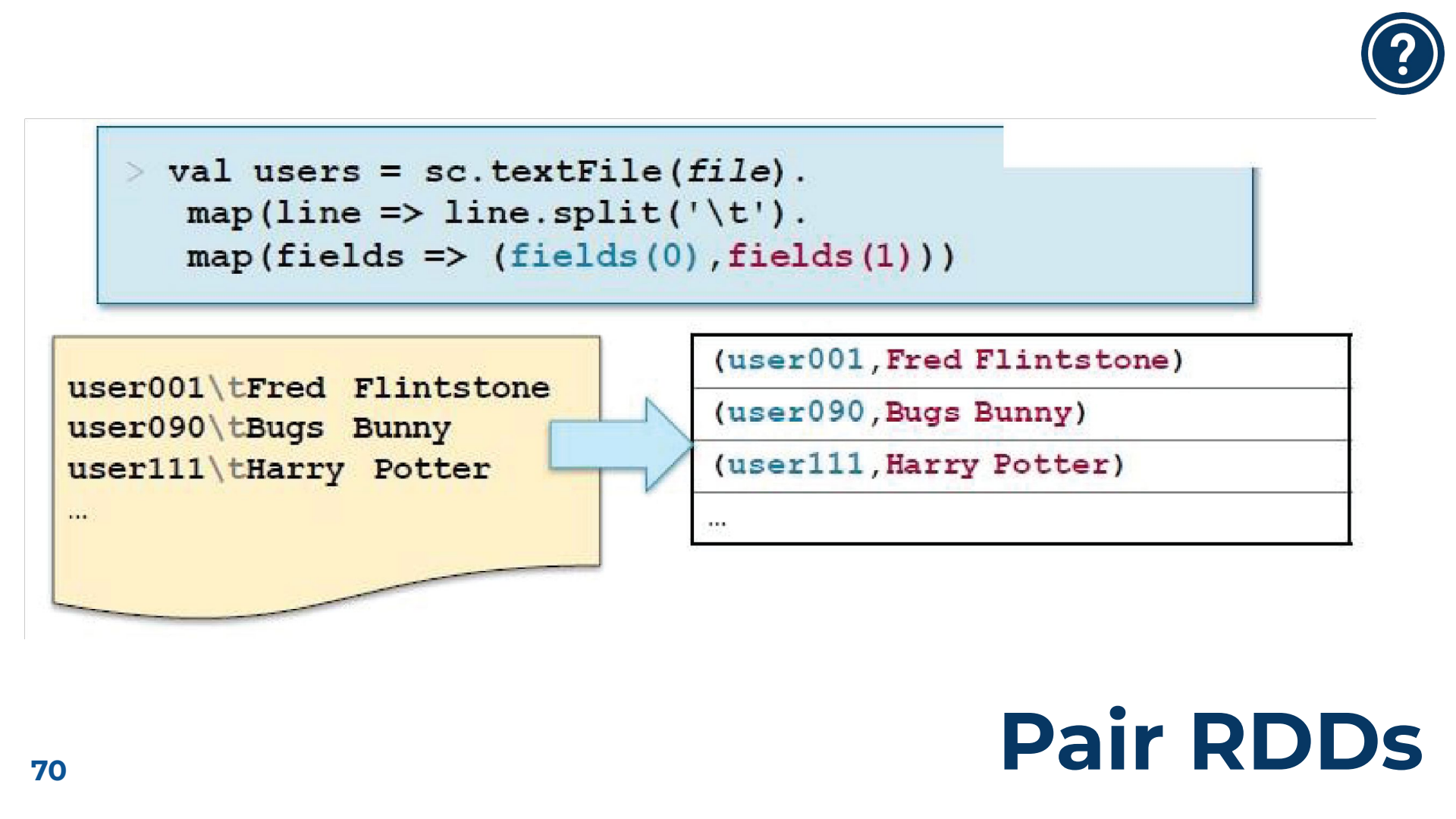
`rdd1.union(rdd2)`

Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

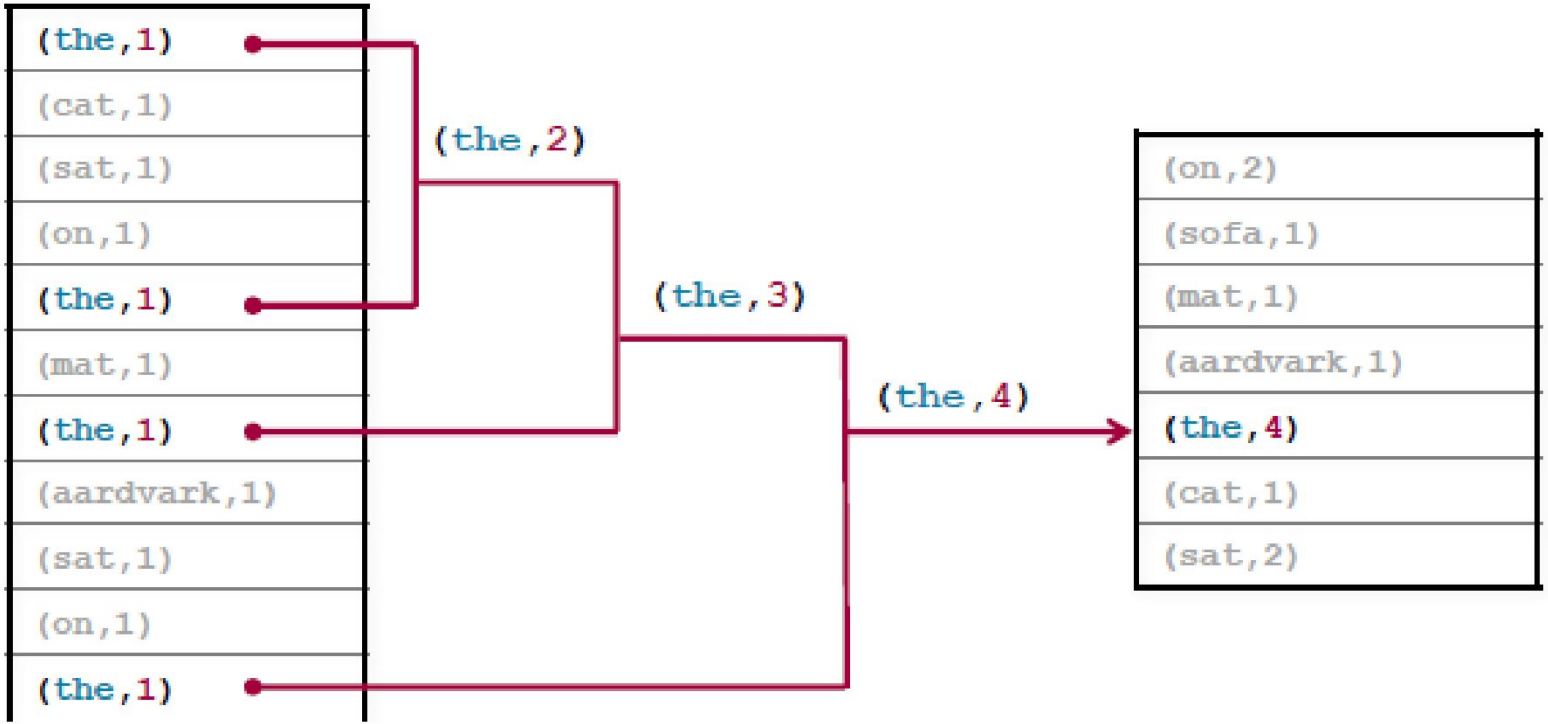
Multi-RDD Operations



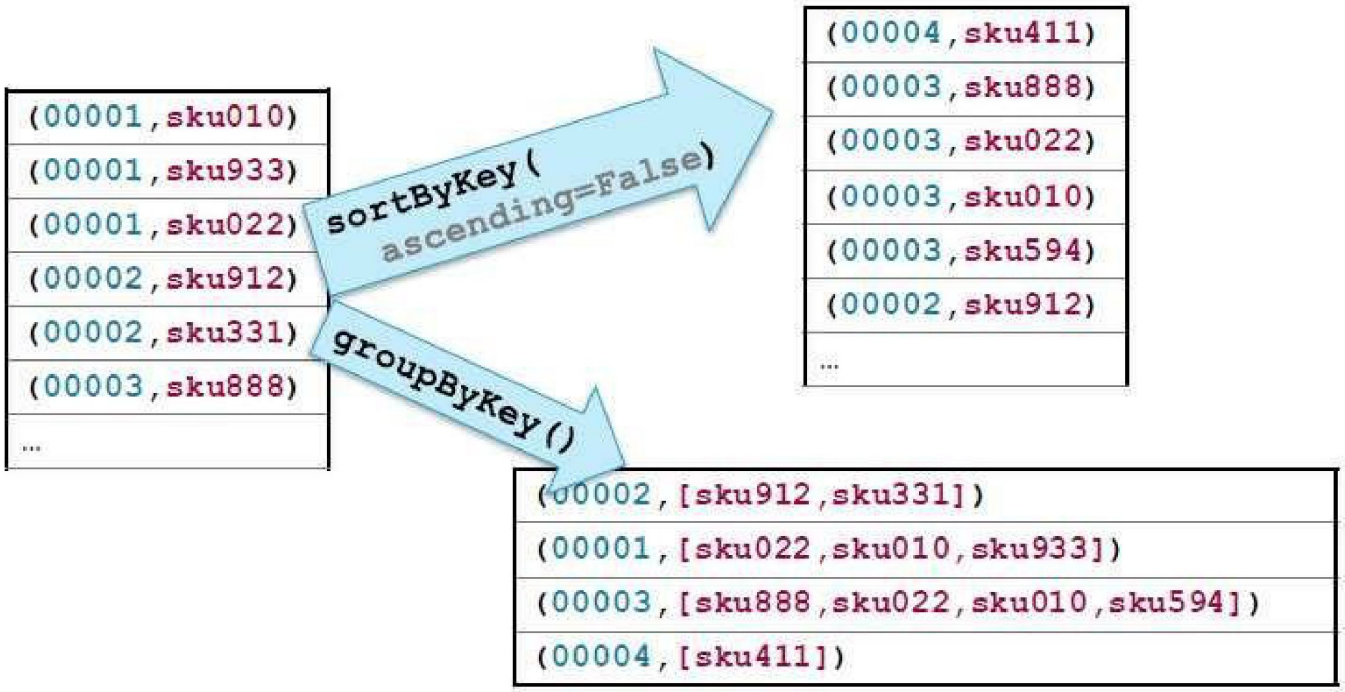
Pair RDDs


> val users = sc.textFile(file).
map(line => line.split('\t').
map(fields => (fields(0),fields(1)))
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…

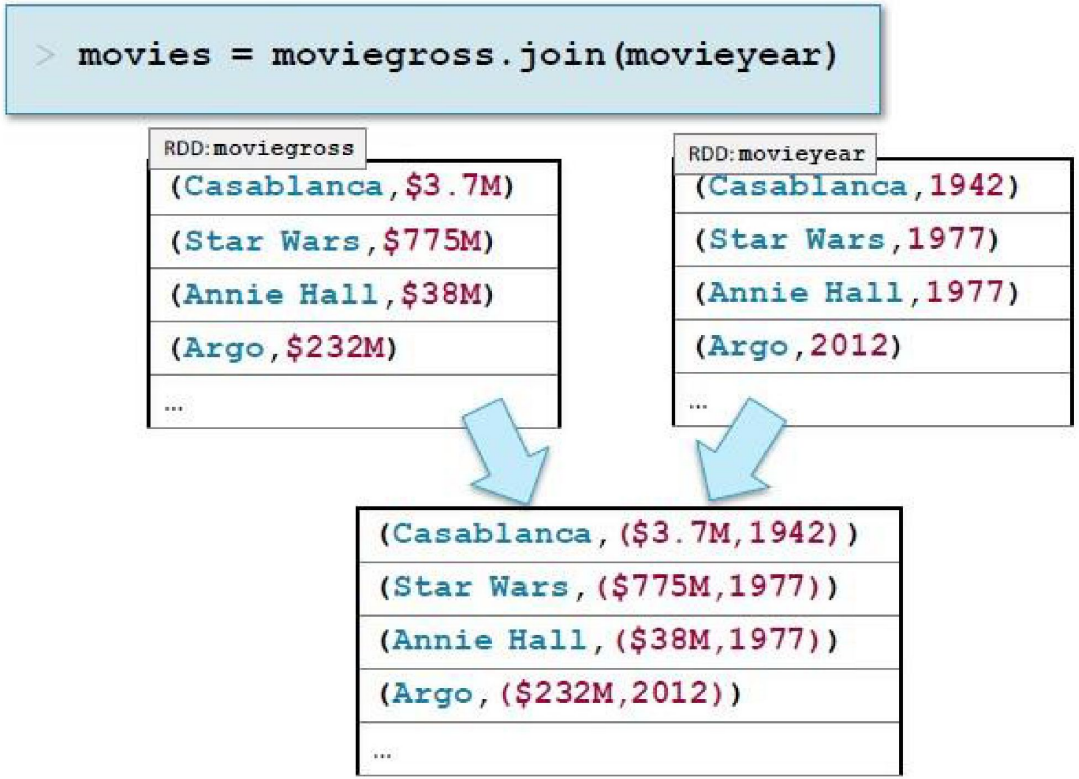
Pair RDDs



reduceByKey



Sorting and grouping



Joining PairRDDs