Introduction to Big Data with Apache Spark

BerkeleyX

This Lecture

Programming Spark

Resilient Distribu

Creating an RD

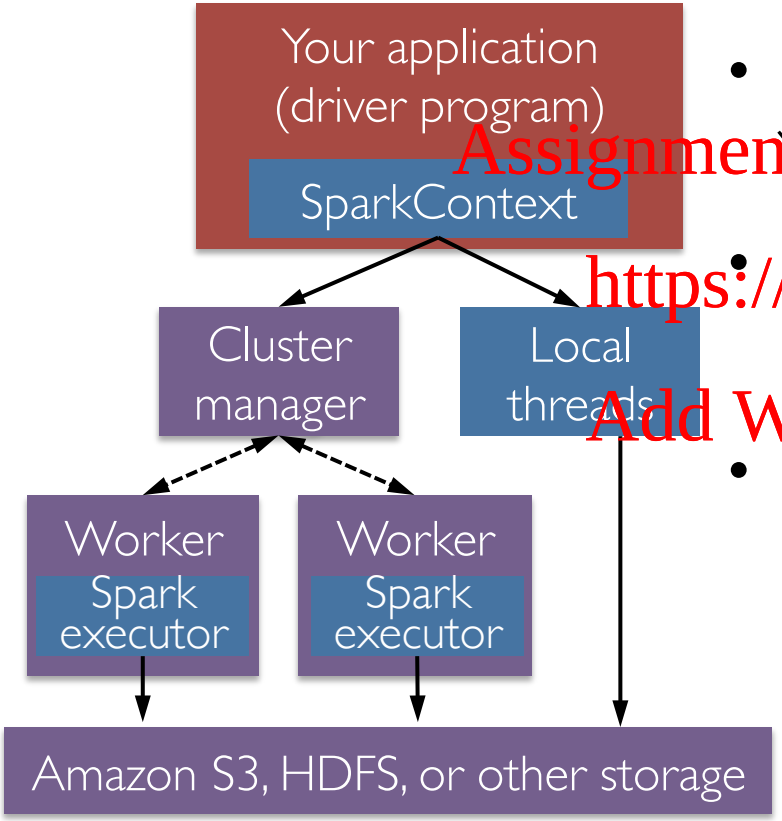
Spark Transformations and Ac

Spark Programming Model

Python Spark (pySpark)

- We are using the Python programming interface to Spark (pySpark)
- pySpark provides an easy-to-use programming abstraction and parallel runtime:
  - » "Here's an operation, run it on all of the data"
- RDDs are the key concept

Spark Driver and Workers



- A Spark program is two programs:
  - A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers

Spark Context

- A Spark program first creates a **`SparkContext`** object
  - » Tells Spark how and where to access a cluster
  - » pySpark shell and Data… …te the **`sc`** variable
  - » iPython and programs must use a construc… …new **`SparkContext`**
- Use **`SparkContext`** to create RD…

In the labs, we create the SparkContext for you

Spark Essentials: Master

- The **`master`** parameter for a **`SparkContext`** determines which type and size of cluster to use

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

In the labs, we set the master parameter for you

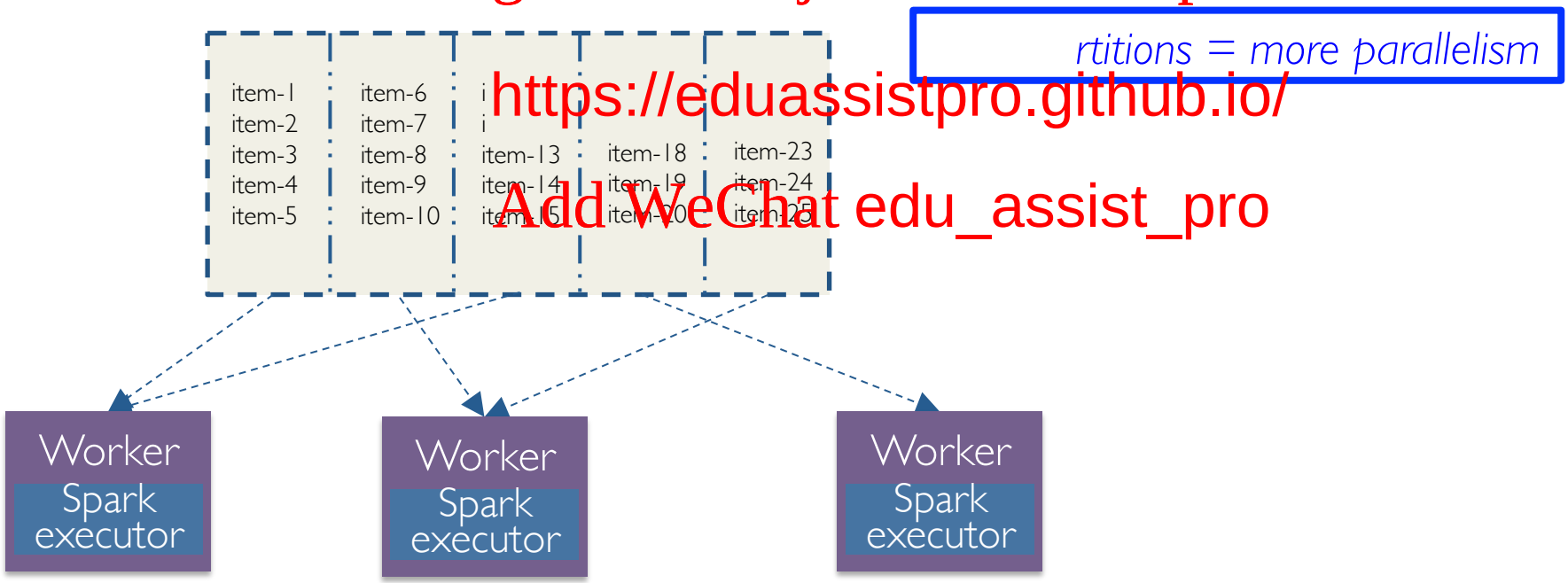
Resilient Distributed Datasets

- The primary abstraction in Spark
  - » Immutable once constructed
  - » Track lineage information to efficiently recompute lost data
  - » Enable operations on collection of elements in parallel
- You construct RDDs
  - » by *parallelizing* existing Python collections (lists)
  - » by *transforming* an existing RDDs
  - » from *files* in HDFS or any other storage system

RDDs

- Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

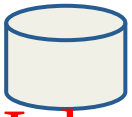


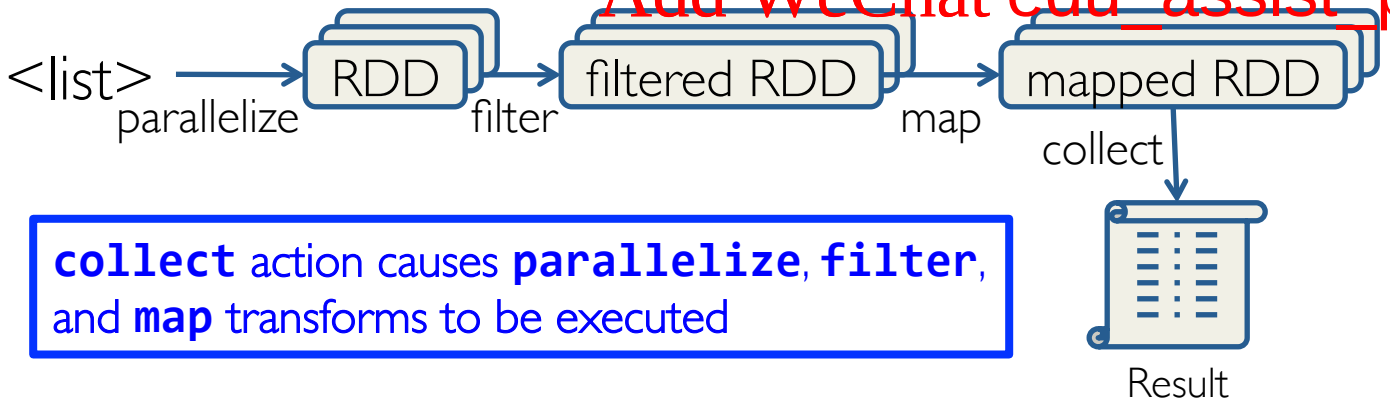
rtitions = more parallelism

RDDs

- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformed R action runs on it
- Persist (cache) RDDs in me

Working with RDDs

- Create an RDD from a data source:  <list>
- Apply transfor map filter
- Apply actions unt



`collect` action causes **`parallelize`**, **`filter`**, and **`map`** transforms to be executed

Spark References

- http://spark.apache.org/docs/latest/programming-guide.html
- http://spark.apache.org/docs/latest/api/python/index.html

Creating an RDD

- Create RDDs from Python collections (lists)

No computation occurs with `sc.parallelize()`

ow to create the RDD with


```
>>> data = [1, 2, 3, 4,

>>> data

[1, 2, 3, 4, 5]


>>> rDD = sc.parallelize(data, 4)

>>> rDD

ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

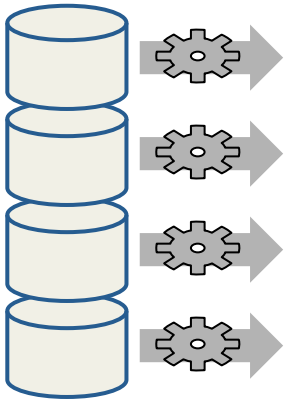
Creating RDDs

- From HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop `InputFormat`, and directory or glob wildcard: `/data/201404*`


```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
   NativeMethodAccessorImpl.java:-2
```

Creating an RDD from a File

```
distFile = sc.textFile("...", 4)
```

- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

Spark Transformations

- Create new datasets from an existing one
- Use *lazy evaluation*: results not computed right away – instead Spark re rmations applied to base dataset
  - » Spark optimizes the required calcul
  - » Spark recovers from failures and slow workers
- Think of this as a recipe for creating result

Some Transformations

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing eac *unc*
`filter(`*`func`*`)`	ret elements of the source on whi
`distinct([`*`numTasks`*`]))`	retu a new dataset that cont elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

Review: Python `lambda` Functions

- Small anonymous functions (not bound to a name)

```
lambda a, b: a + b
```

  - » returns the sum

- Can use lambda functions nction objects are required

- Restricted to a single expression

Transformations

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(la
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4          )
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures automatically passed to workers

Transformations

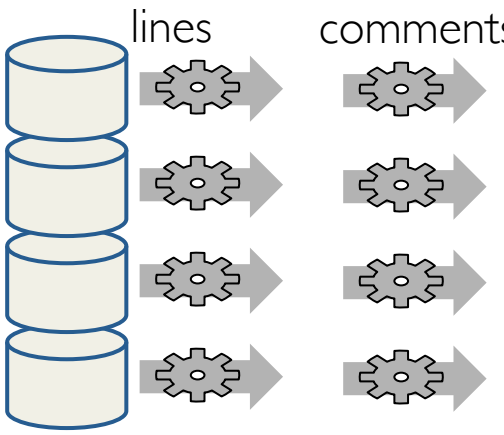
```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(l
RDD: [1, 2, 3] → [1, 6, 2, 7, 3
```


Function literals (green) are closures automatically passed to workers

Transforming an RDD

```
lines = sc.textFile("...", 4)
comments = l                mment                )
```

Lazy evaluati
nothing executes –
Spark saves recipe for
transforming source

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism f Spark

Some Actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is be
`take(`*`n`*`)`	return an array with the fi
`collect()`	return all the elements as WARNING: make sure w m
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

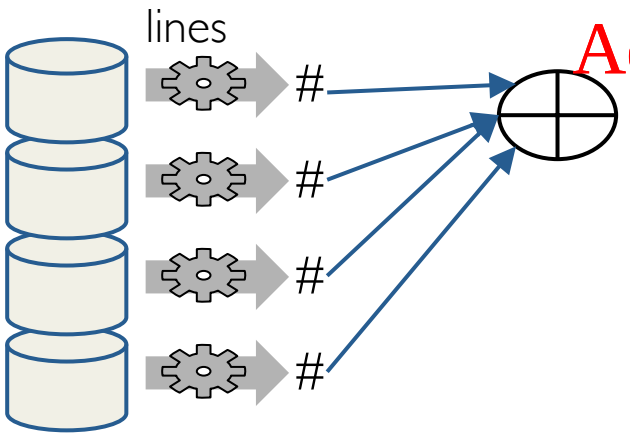
Getting Data Out of RDDs

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] #
```

Spark Programming Model

```
lines = sc.textFile("...", 4)

print lines.
```

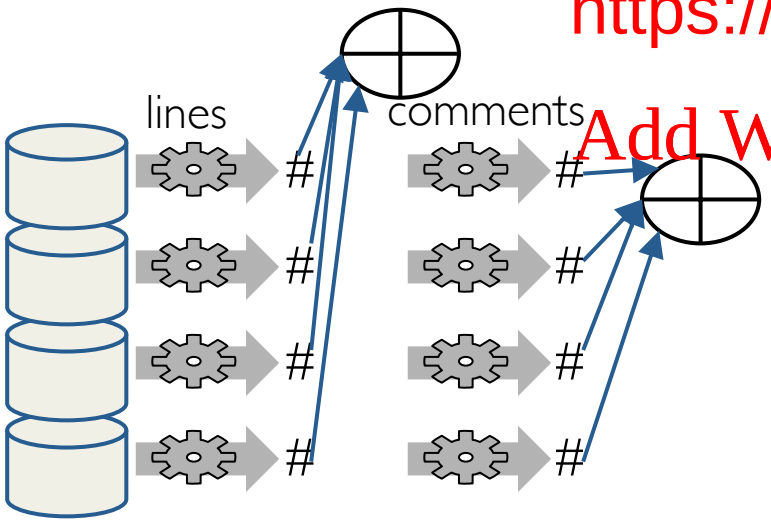



count()

- read d
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.  s.                count()
```

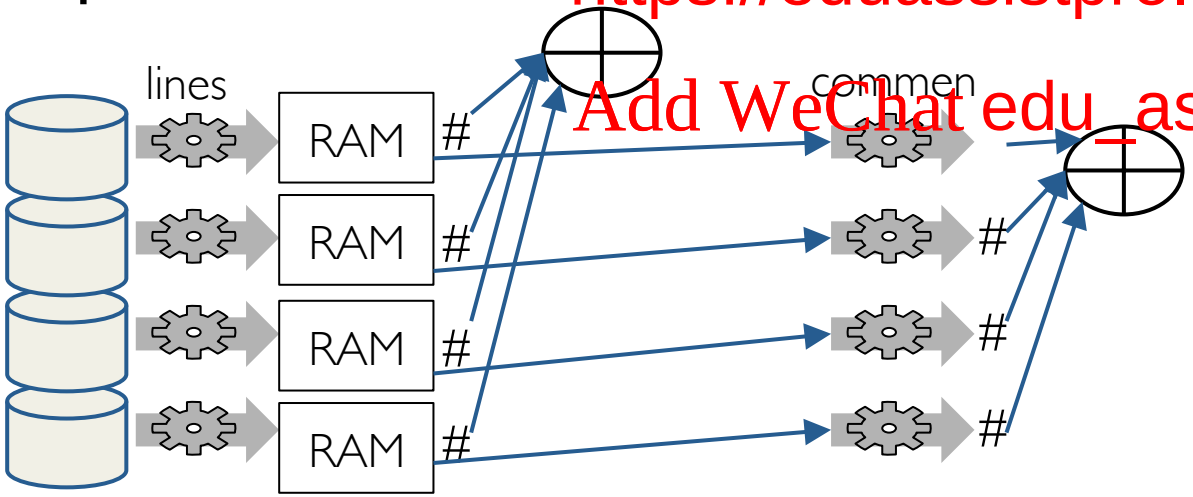



Spark re :

- read
- sum within partitions
- combine sums in driver

Caching RDDs

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count()
```

Spark Program Lifecycle

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. **`cache()`** some RDDs for reuse
4. Perform actions to execute parallel computation and produce results

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element r tuple


```
>>> rdd = sc.parallelize([(1, 2),
RDD: [(1, 2), (3, 4)]
```

Some Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where th giv (V
`sortByKey()`	return a new dataset (K,V) p ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
)]  RDD: [(1,2), (3,4
```



```
>>> rdd2 = sc.parallelize([(1,'          '), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')
                [(1,'a'), (1,'b'), (2,'c')]
```

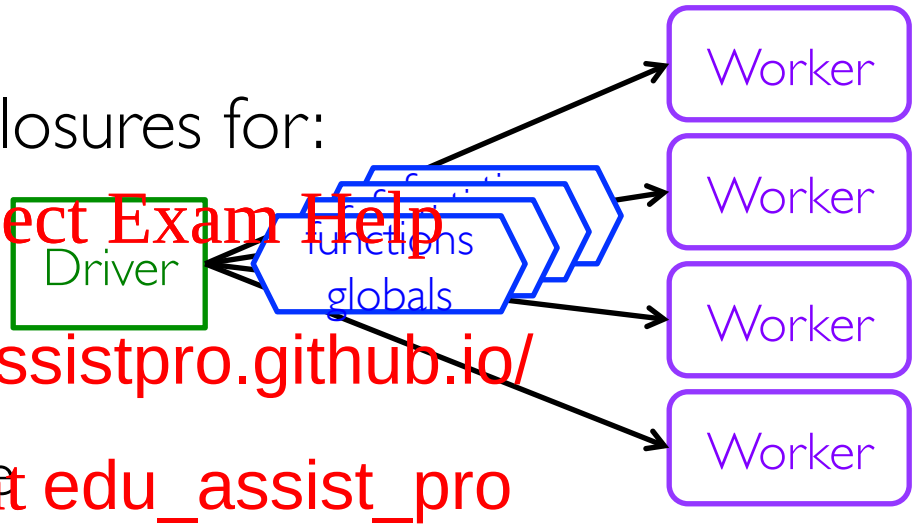
Key-Value Transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1
                [(1,
```


Be careful using **groupByK**
it can cause a lot of data movement across the network and create large Iterables at workers

pySpark Closures

- Spark automatically creates closures for:
  - » Functions that run
  - » Any global variables used by those



- One closure per worker
  - » Sent for ***every*** task
  - » No communication between workers
  - » Changes to global variables at workers are not sent to driver

Consider These Use Cases

- Iterative or single jobs with large global variables
  - » Sending large read-only lookup table to workers
  - » Sending large feature vector in a ML algorithm to workers
- Counting even execution
  - » How many input lines were blank
  - » How many input records were co

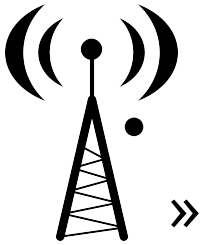
Consider These Use Cases

- Iterative or single jobs with large global variables
  - » Sending large read-only lookup table to workers
  - » Sending large feature vector in a ML algorithm to workers
- Counting even execution
  - » How many input lines were blank
  - » How many input records were co

Problems:

- Closures are (re-)sent with *every* job
- Inefficient to send large data to each worker
- Closures are one way: driver ➔ worker

pySpark Shared Variables

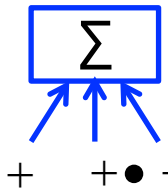


Broadcast Variables

» Efficiently send large, *read-only* value to all workers

» Saved at workers for use in one or more Spark operations

» Like sending a l e to all the nodes



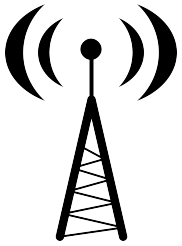
Accumulators

» Aggregate values from workers back to driver

» Only driver can access value of accumulator

» For tasks, accumulators are write-only

» Use to count errors seen in RDD across workers

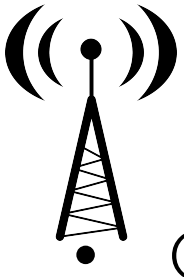


Broadcast Variables

- Keep *read-only* variable cached on workers
  - » Ship to each worker only once instead of with each task
- Example: effi r a large dataset
- Usually distrib adcast algorithms


```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At a worker (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```



Broadcast Variables Example

Country code lookup for HAM radio call signs

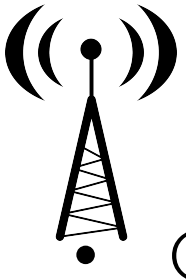
```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country co
signPrefixes = loadCallS

def processSignCount(sign_count, signPrefi
    country = lookupCountry(sign_count[0],
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Expensive to send large table
(Re-)sent for every processed file

From: http://shop.oreilly.com/product/0636920028512.do



Broadcast Variables Example

Country code lookup for HAM radio call signs

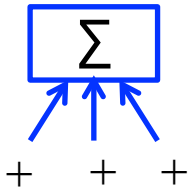
```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country co
signPrefixes = sc.broadc

def processSignCount(sign_count, signPrefi
    country = lookupCountry(sign_count[0],
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```


fficiently sent once to workers

From: http://shop.oreilly.com/product/0636920028512.do

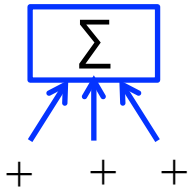


Accumulators

- Variables that can only be "added" to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can value, not tasks


```
>>> accum = sc.accum
>>> rdd = sc.parallelize([1, 2, 3, 4
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```



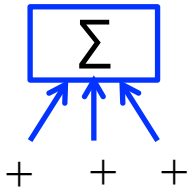
Accumulators Example

- Counting empty lines


```
file = sc.textFile(inputFile)
# Create Accumulator[In
blankLines = sc.accumul

def extractCallSigns(line):
    global blankLines # Make the global v      sible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```



Accumulators

- Tasks at workers cannot access accumulator's values
- Tasks see accumulators as write-only variables
- Accumulators or transformations:
  - » Actions: each ta r is *applied only once*
  - » Transformations: *no guarantees* debugging)
- Types: integers, double, long, float
  - » See lab for example of custom type

Summary