Spark for Scientific Computing

A. Zonca, M. Tatineni - SDSC

What is Spark?

- A distributed computing framework

Problem 1: Storage

- Big data
- Commodity hardware (Cloud)

Solution: Distributed File System

- redundant
- fault tolerant

Problem 2: Computation

- Slow to move data across network
- Computations fail

Solution: Hadoop Mapreduce / Spark

- Execute computation where data are located
- Rerun failed jobs

Problem 3: Communication

- Most of the times, need to summarize data to get a result
- Reduction phase in MapReduce
- Need data transfer across network

Solution: highly optimized Shuffle (All-to-All)

Spark and Hadoop

- Works within the Hadoop ecosystem
- Extends MapReduce
- Initially developed at UC Berkeley
- Now within the Apache Foundation
- ~400 and more developers

Key features of Spark

- **Resiliency**: tolerant to node failures
- **Speed**: supports in-memory caching
- **Ease of use**:
  - Python/Scala interfaces
  - interactive shells
  - many distributed primitives

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Spark 100TB benchmark

HPC: Distributed TBs of data

- Fault-tolerant batch processing
- Data exploration with an interactive console
- SQL operations with Spark-SQL
- Iterative Machine Learning algorithms with Spark-MLlib

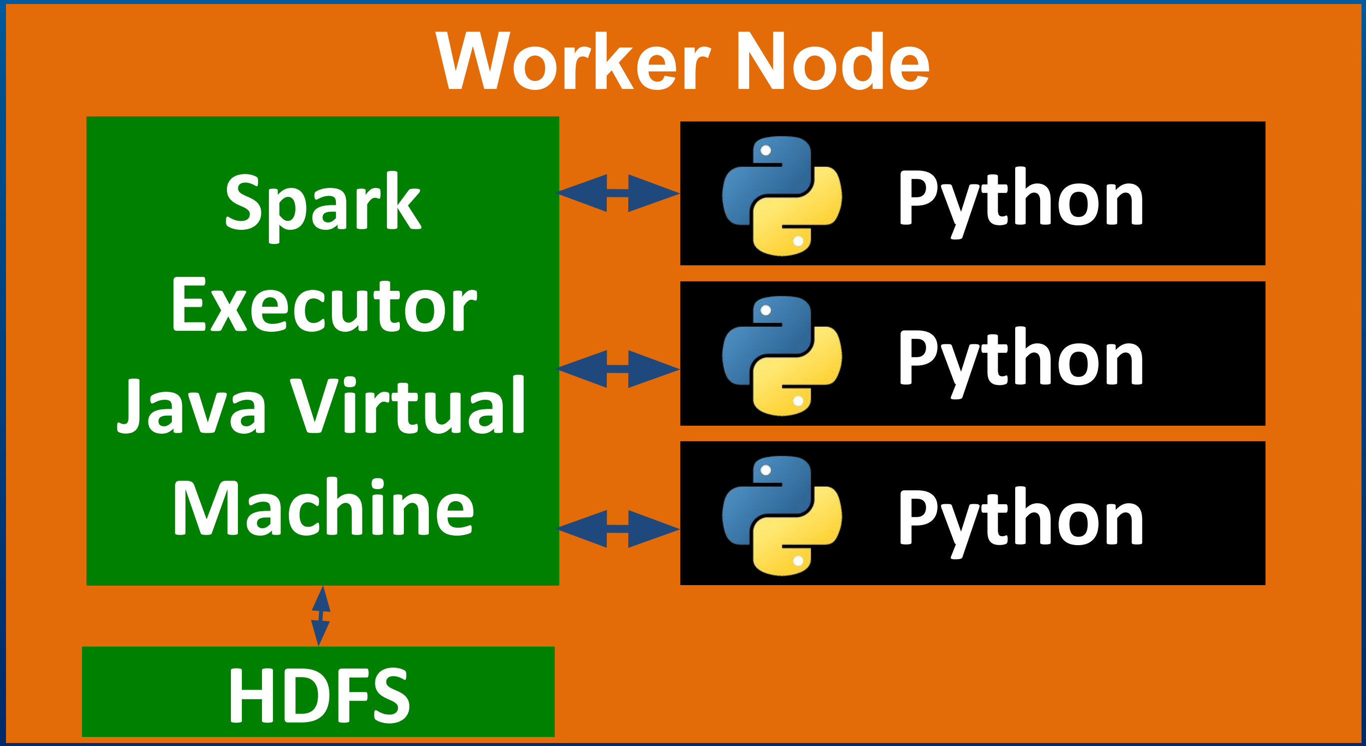
Comparison with MPI

- MPI: describe computation and communication explicitly
- Spark: use a graph of high-level operators, the framework decides how and where to run tasks

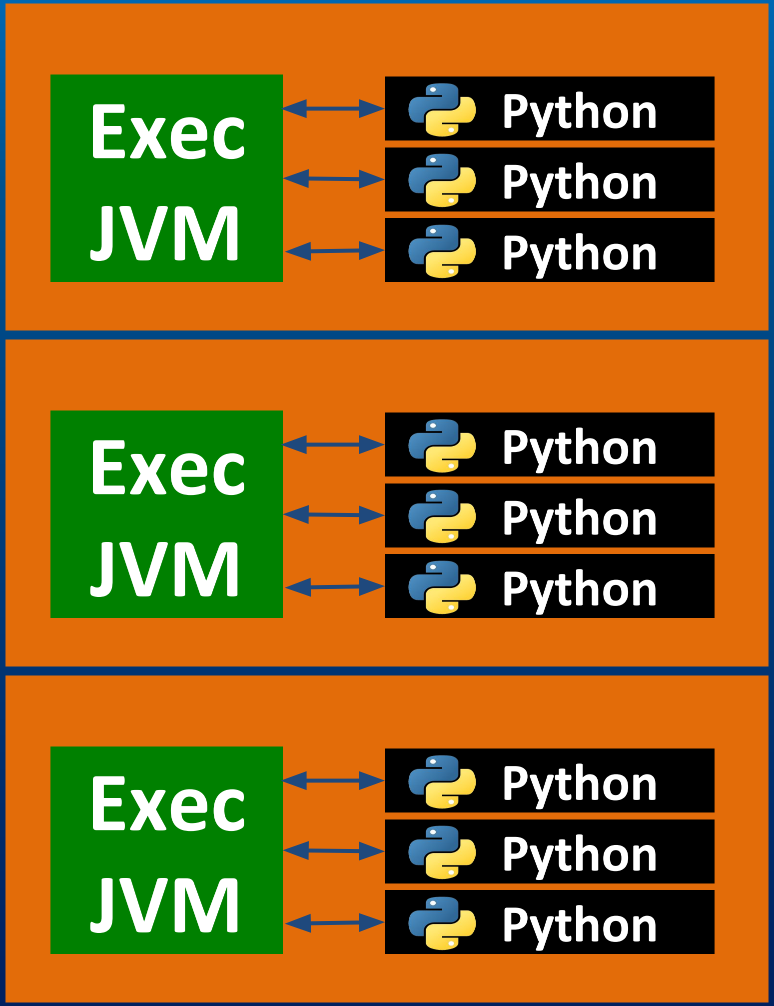
Interactive spark on Comet

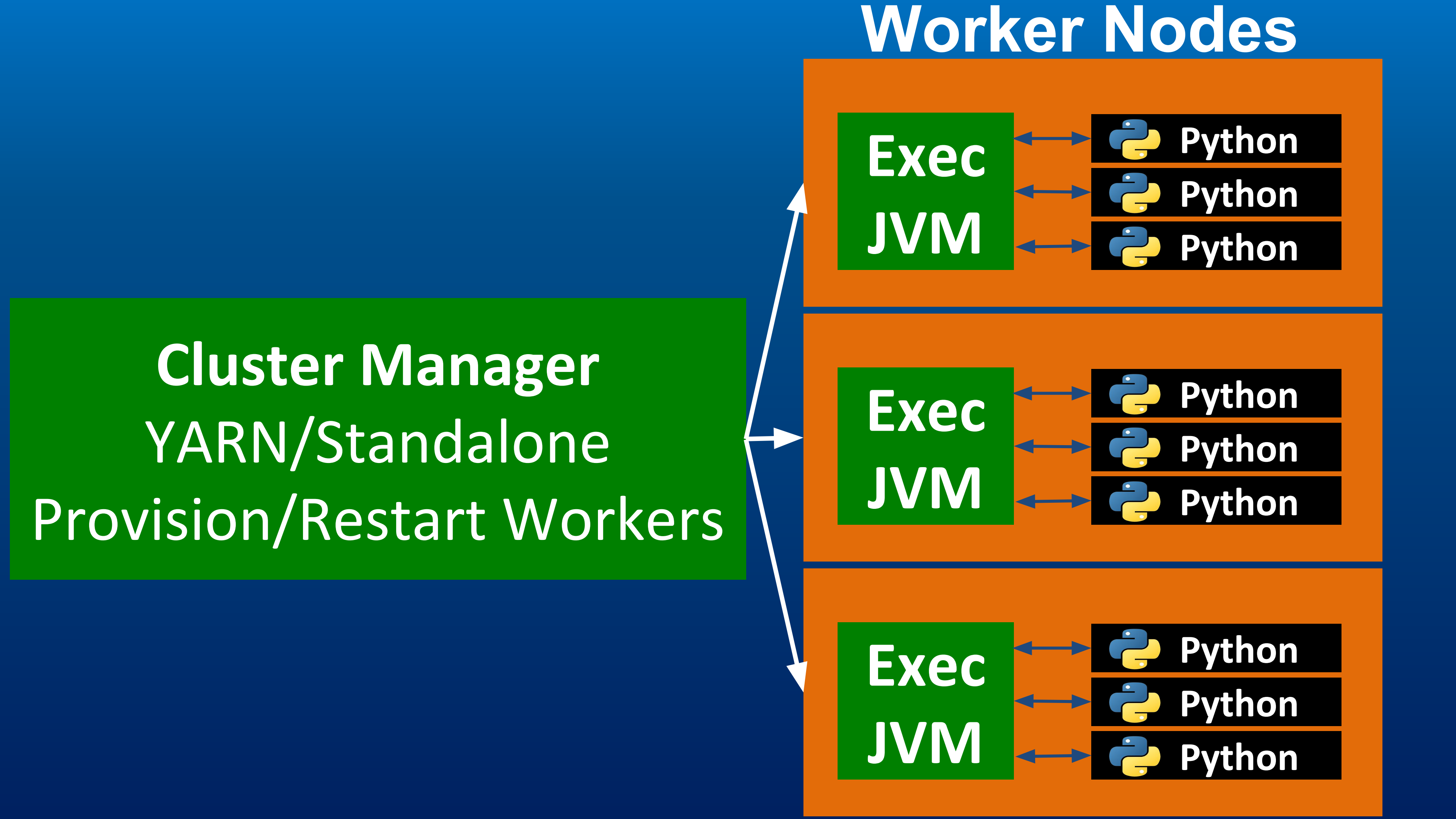
Add to .bashrc:
module load python scipy

Navigate to the spark/ folder,
submit a spark job with:
> qsub spark.cmd

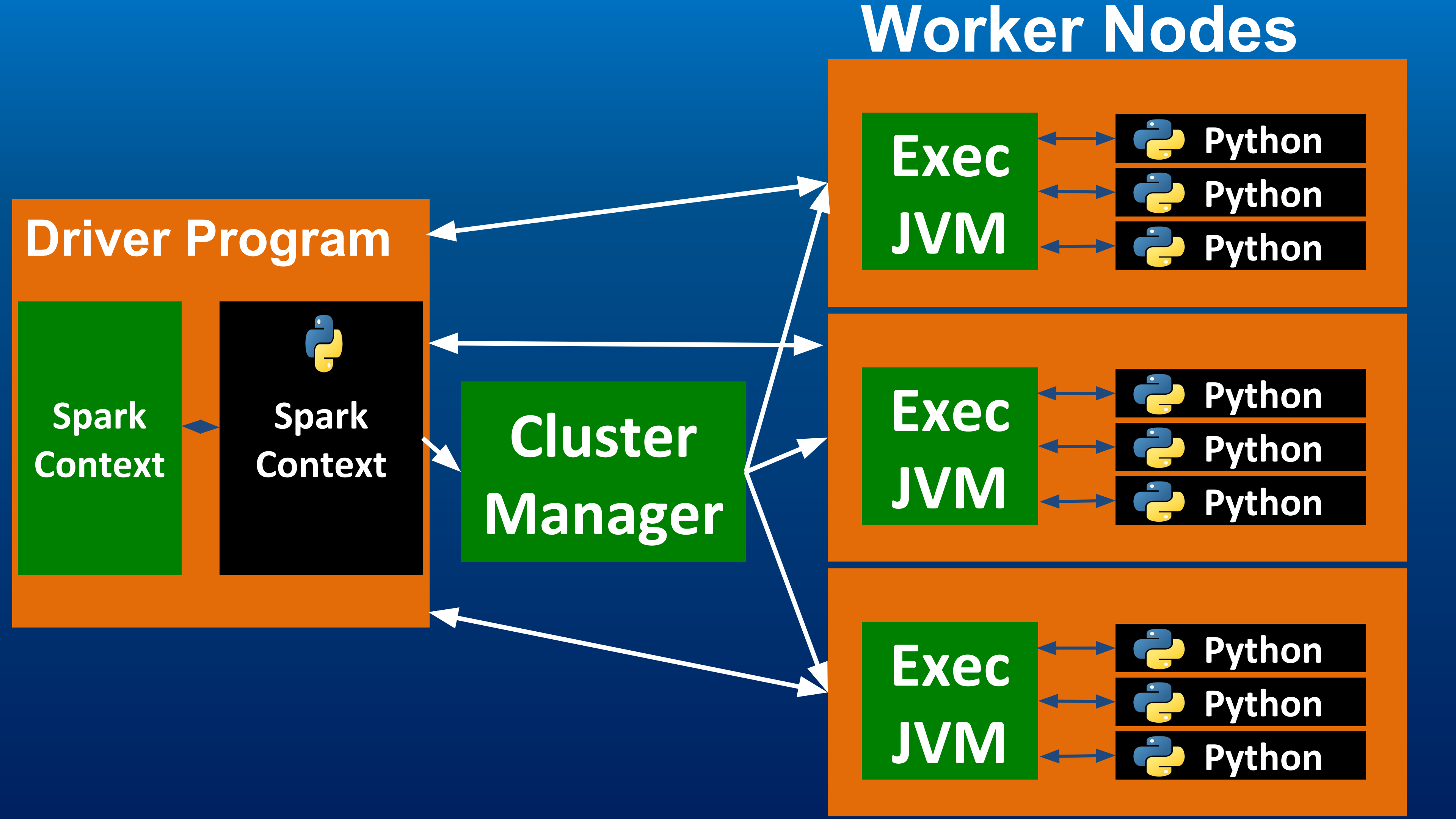

Worker Node
Spark Executor Java Virtual Machine
Python
Python
Python
HDFS

Worker Nodes

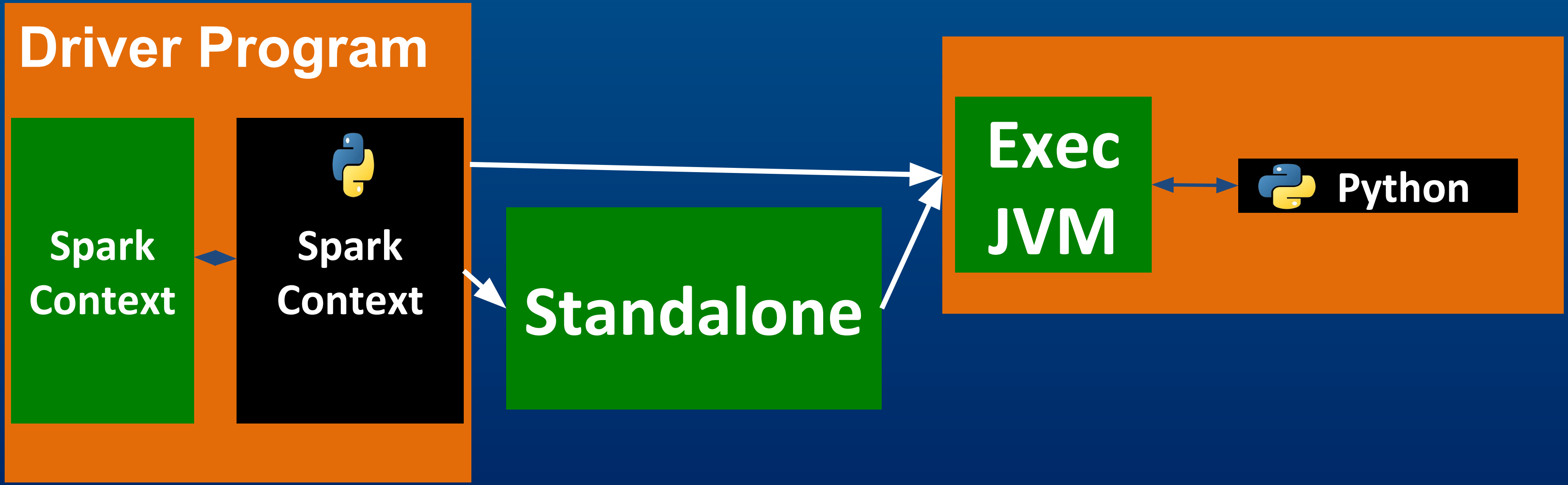




Worker Nodes
Cluster Manager
YARN/Standalone
Provision/Restart Workers
Exec
JVM
Python
Python
Python
Exec
JVM
Python
Python
Python
Exec
JVM
Python
Python
Python

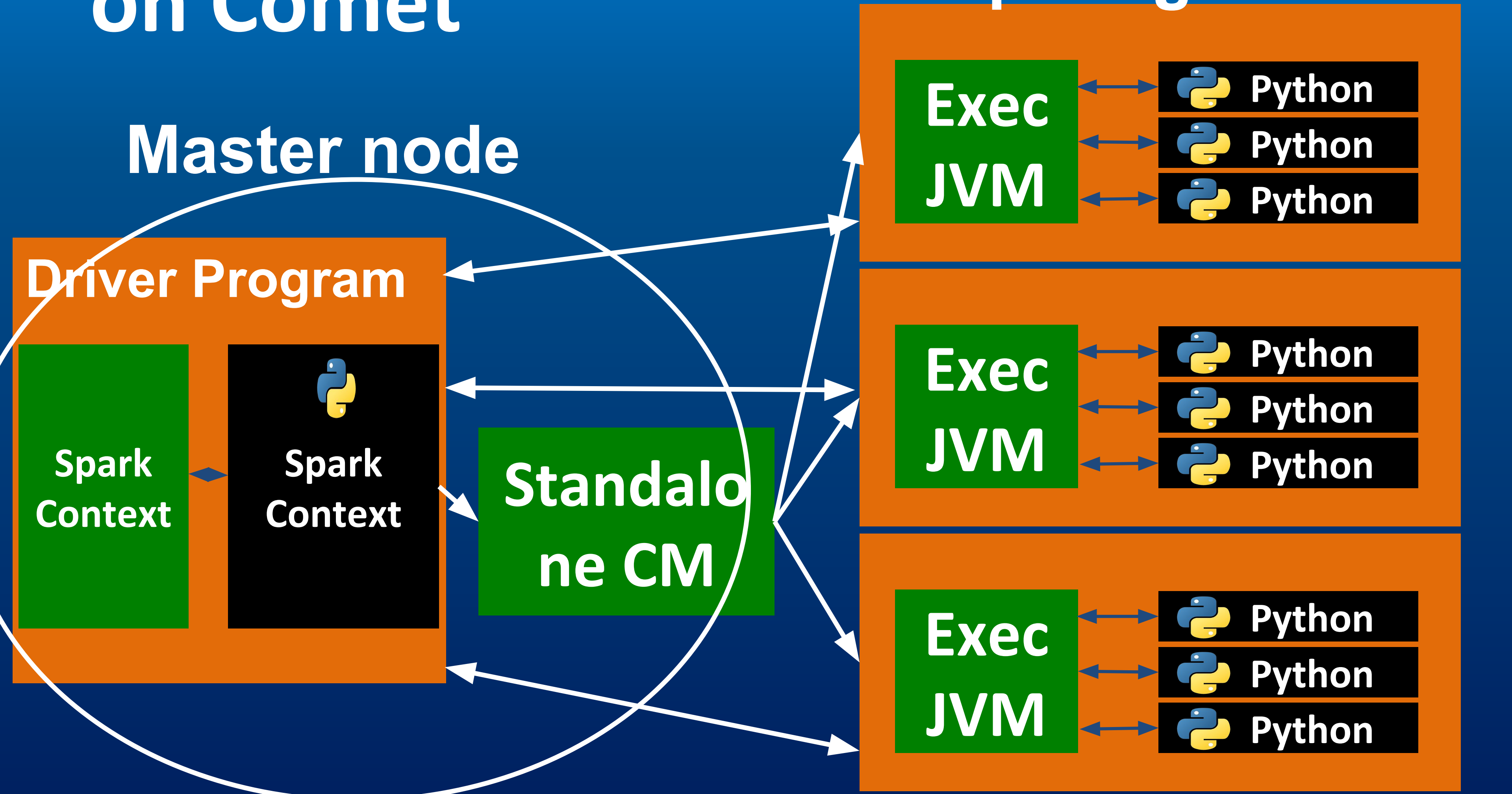

Worker Nodes
Driver Program
Spark Context
Spark Context
Cluster Manager
Exec JVM
Python
Python
Python
Exec JVM
Python
Python
Python
Exec JVM
Python
Python
Python

Spark Local

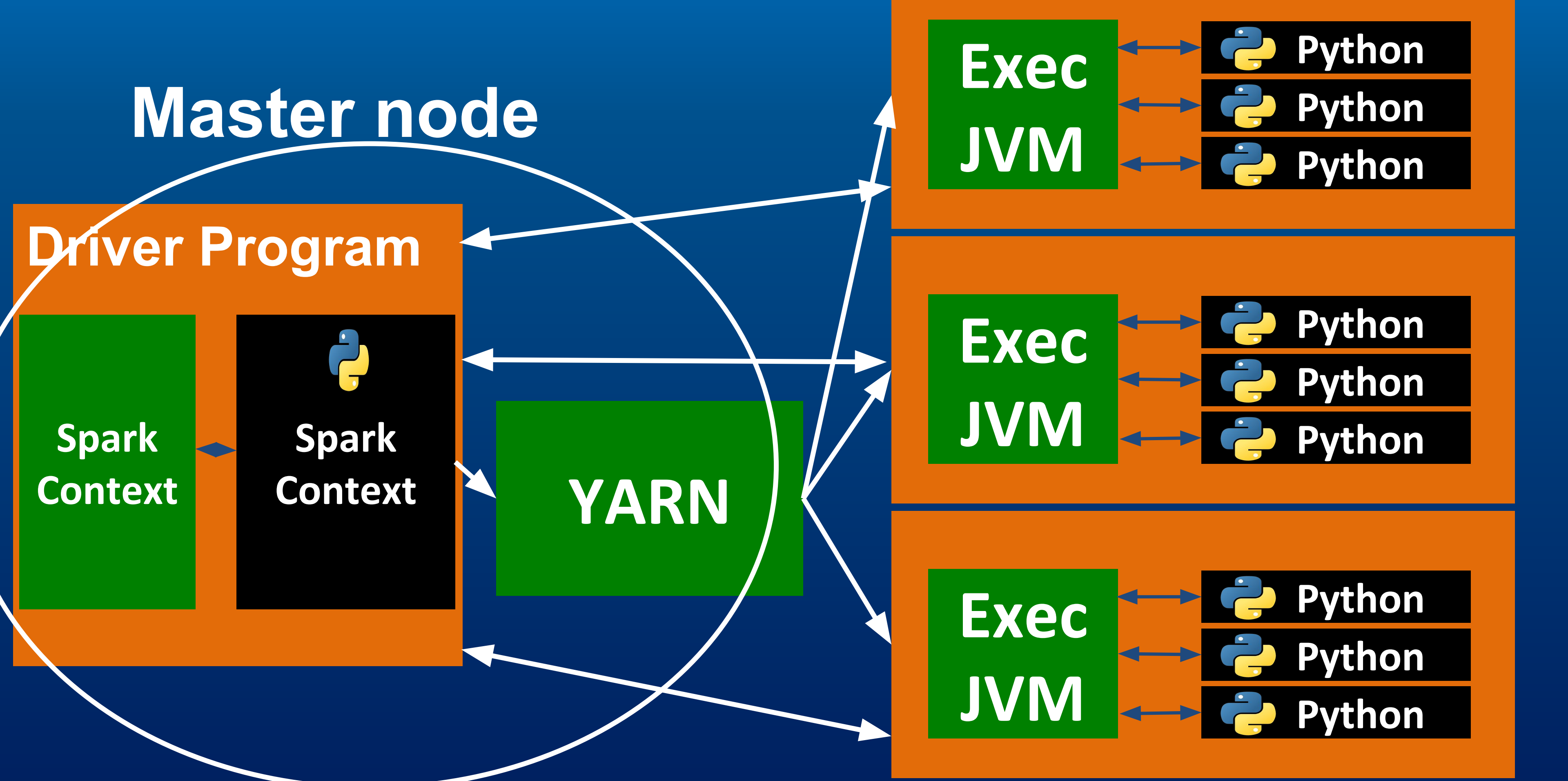

Driver Program
Spark Context
Spark Context
Standalone
Exec JVM
Python

on Comet

Computing Nodes



on Amazon EMR



Connect to Spark

```
source slurm-env.sh
ssh $SLURMD_NODENAME

source slurm-env.sh
pyspark
```

First example

```
local_data = range(100)
data = sc.parallelize(local_data)
def myfilter(d):
    return d < 10
data.glom().collect()
data.filter(myfilter).collect()
```

Read text into Spark

from local filesystem:

```
text_RDD = sc.textFile("file:
///home/USER/workshop/spark/starwars.txt")
 text_RDD.take(1) #outputs the first line
```

Wordcount in Spark: map

```
def split_words(line):
    return line.split()

def create_pair(word):
    return (word, 1)

pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

```
pairs_RDD.collect()
Out[]: [(u'A', 1),
 (u'long', 1),
 (u'time', 1),
 (u'ago', 1),
 (u'in', 1),
 (u'a', 1),
 (u'galaxy', 1),
 (u'far', 1),
 (u'far', 1),
 (u'away', 1)]
```

Wordcount in Spark: reduce

```
def sum_counts(a, b):
    return a + b

wordcounts_RDD = pairs_RDD.reduceByKey(sum_counts)

wordcounts_RDD.collect()
```

```
Out[]:
[(u'A', 1),
 (u'ago', 1),
 (u'far', 2),
 (u'away', 1),
 (u'in', 1),
 (u'long', 1),
 (u'a', 1),
 (u'time', 1),
 (u'galaxy', 1)]
```

Resilient Distributed Dataset

- Resilient: fault tolerant, lineage is saved, lost partitions can be recovered
- Distributed: partitions are automatically distributed across nodes
- Created from: HDFS, S3, HBase, Local file, Local hierarchy of folders

Read from HDFS

```
$ hdfs dfs -mkdir -p /user/$USER
$ hdfs dfs -put songs.tsv /user/$USER/
```

```
sc.textFile('hdfs:///user/%s/songs.tsv' % os.
environ["USER"])
```

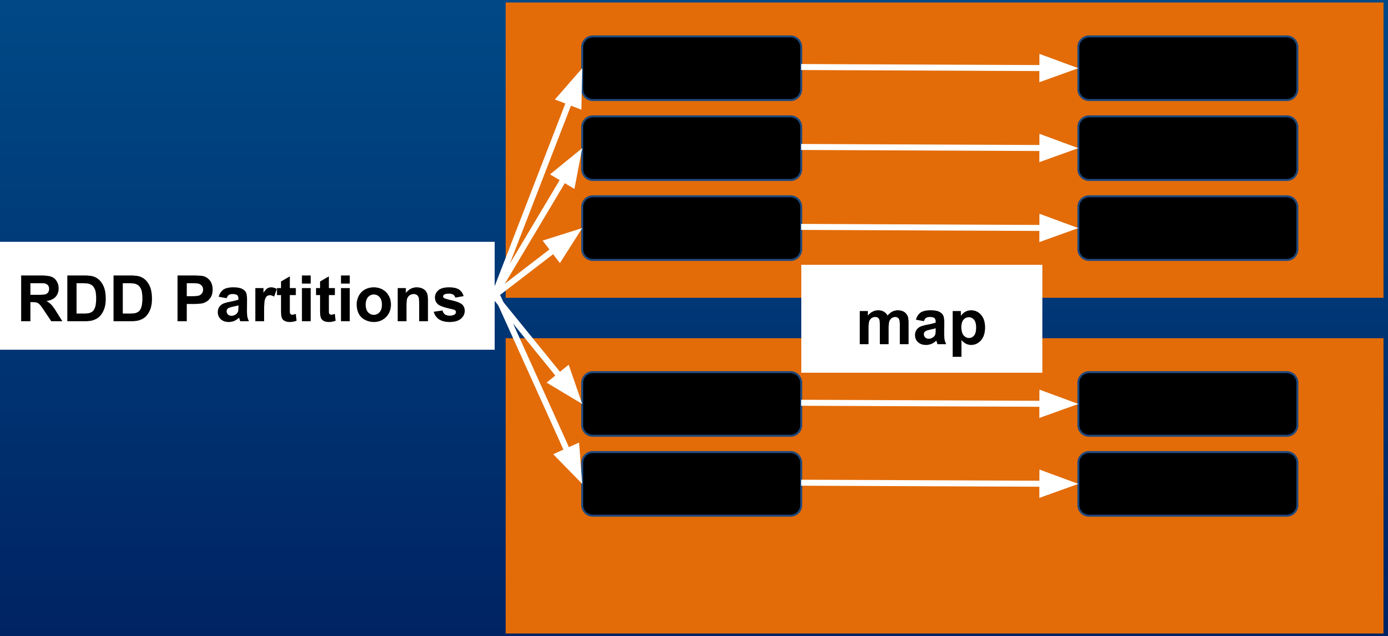
Apply a transformation: map

`map` : apply function to each element of RDD

```
def lower(line):
    return line.lower()
lower_text_RDD = text_RDD.map(lower)
```

map

map : apply function to each element of RDD



Other transformations

- flatMap(func) - map then flatten output
- filter(func) - keep only elements where func is true
- sample(withReplacement, fraction, seed) - get a random data fraction
- coalesce(numPartitions) - merge partitions to reduce them to numPartitions

flatMap

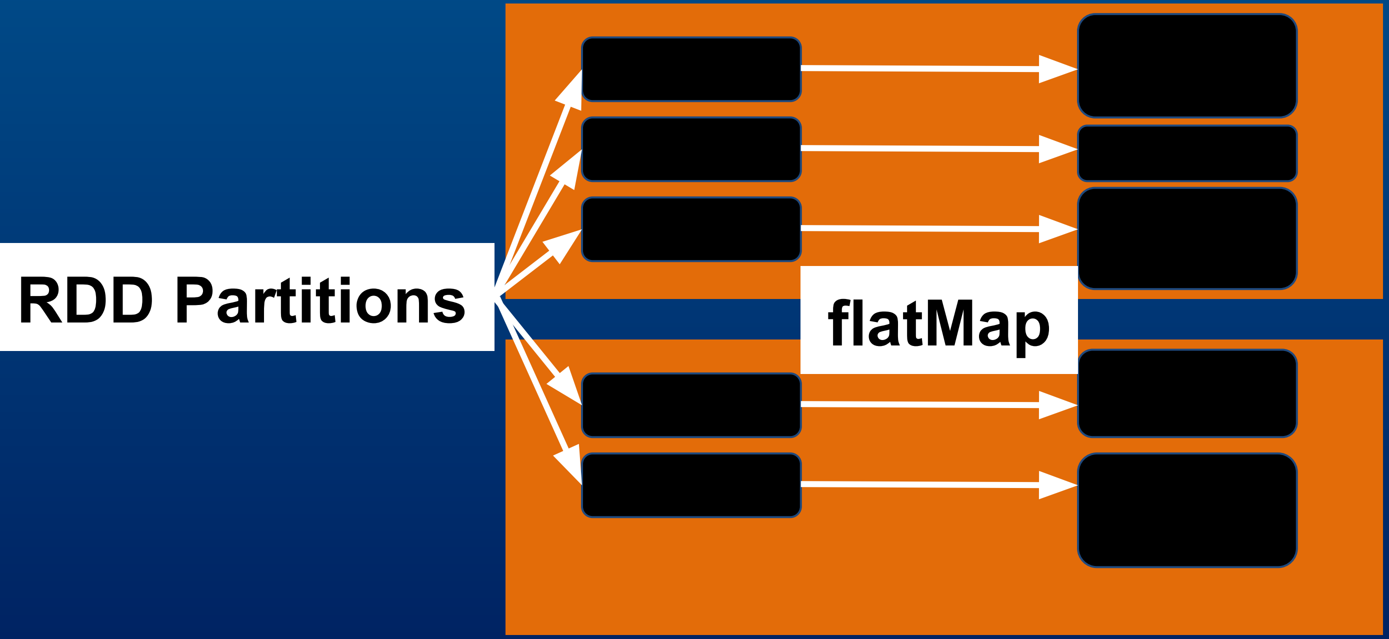
```
def split_words(line):
    return line.split()

words_RDD = text_RDD.
flatMap(split_words)
words_RDD.collect()
```

Out[]:

```
[u'A',
 u'long',
 u'time',
 u'ago',
 u'in',
 u'a',
 u'galaxy',
 u'far',
 u'far',
 u'away']
```

flatMap

flatMap : map then flatten output



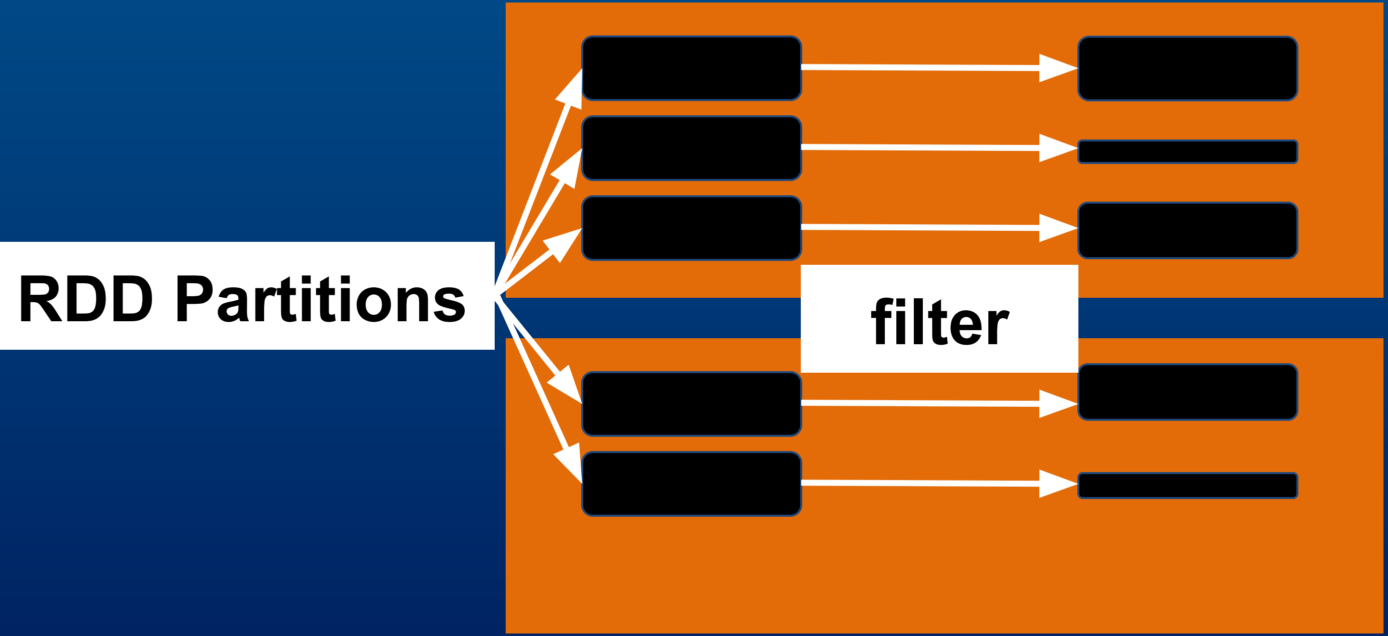
filter

```
def starts_with_a(word):
    return word.lower().startswith("a")

words_RDD.filter(starts_with_a).collect()
```

```
Out[]: [u'A', u'ago', u'a', u'away']
```

filter

`filter` : keep only elements where func is true



coalesce

```
sc.parallelize(range(10), 4).glom().collect()
```

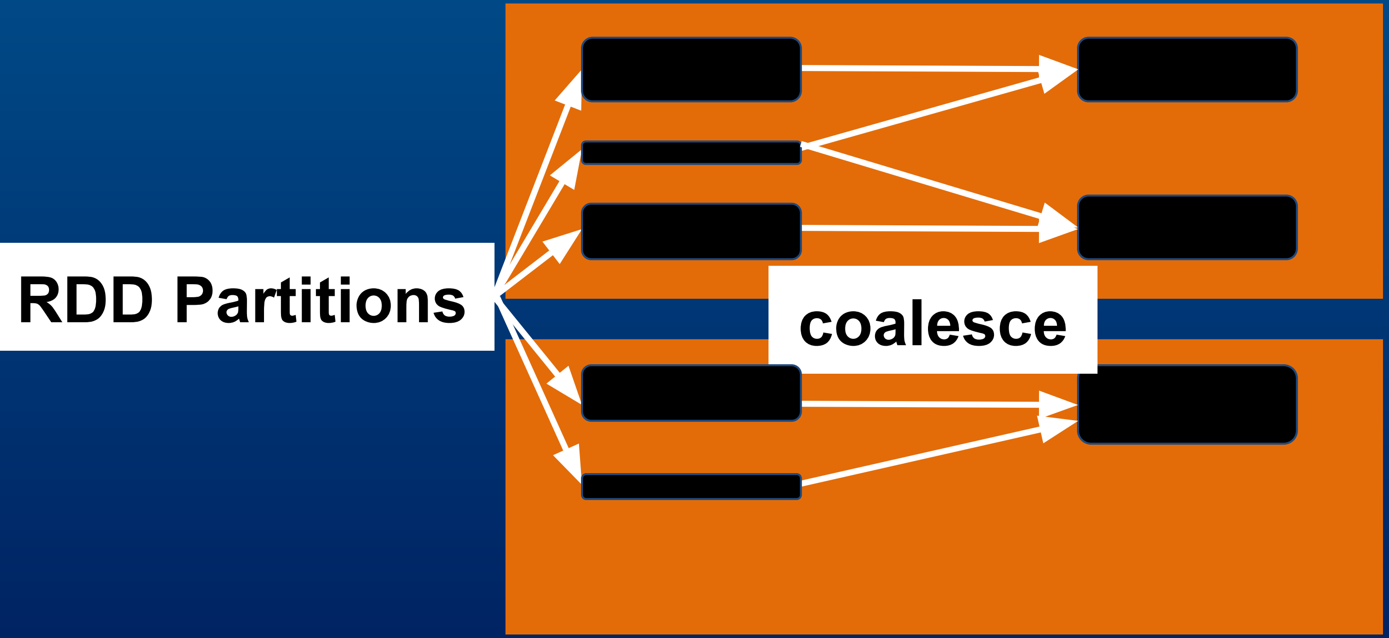
```
Out[]: [[0, 1], [2, 3], [4, 5], [6, 7, 8, 9]]
```

```
sc.parallelize(range(10), 4).coalesce(2).glom().collect()
```

```
Out[]: [[0, 1, 2, 3], [4, 5, 6, 7, 8, 9]]
```

coalesce

coalesce : reduce the number of partitions



Wide Transformations

Transformations of (K,V) pairs

```python
def create_pair(word):
    return (word, 1)

pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

```
pairs_RDD.collect()
```

```
Out[]: [(u'A', 1),
 (u'long', 1),
 (u'time', 1),
 (u'ago', 1),
 (u'in', 1),
 (u'a', 1),
 (u'galaxy', 1),
 (u'far', 1),
 (u'far', 1),
 (u'away', 1)]
```

groupByKey

groupByKey : (K, V) pairs => (K, iterable of all V)

(A, 1)

(B, 8)

(A, 2)

(A, 5)

→

(A, [1, 2, 5])

(B, [8])

```
pairs_RDD.groupByKey().collect()
```

Out[]: [(u'A', <pyspark.resultiterable.ResultIterable at XXX>),

(u'ago', <pyspark.resultiterable.ResultIterable at XXX>),

(u'far', <pyspark.resultiterable.ResultIterable at XXX>),

(u'away', <pyspark.resultiterable.ResultIterable at XXX>),

(u'in', <pyspark.resultiterable.ResultIterable at XXX>),

(u'long', <pyspark.resultiterable.ResultIterable at XXX>),

(u'a', <pyspark.resultiterable.ResultIterable at XXX>),<

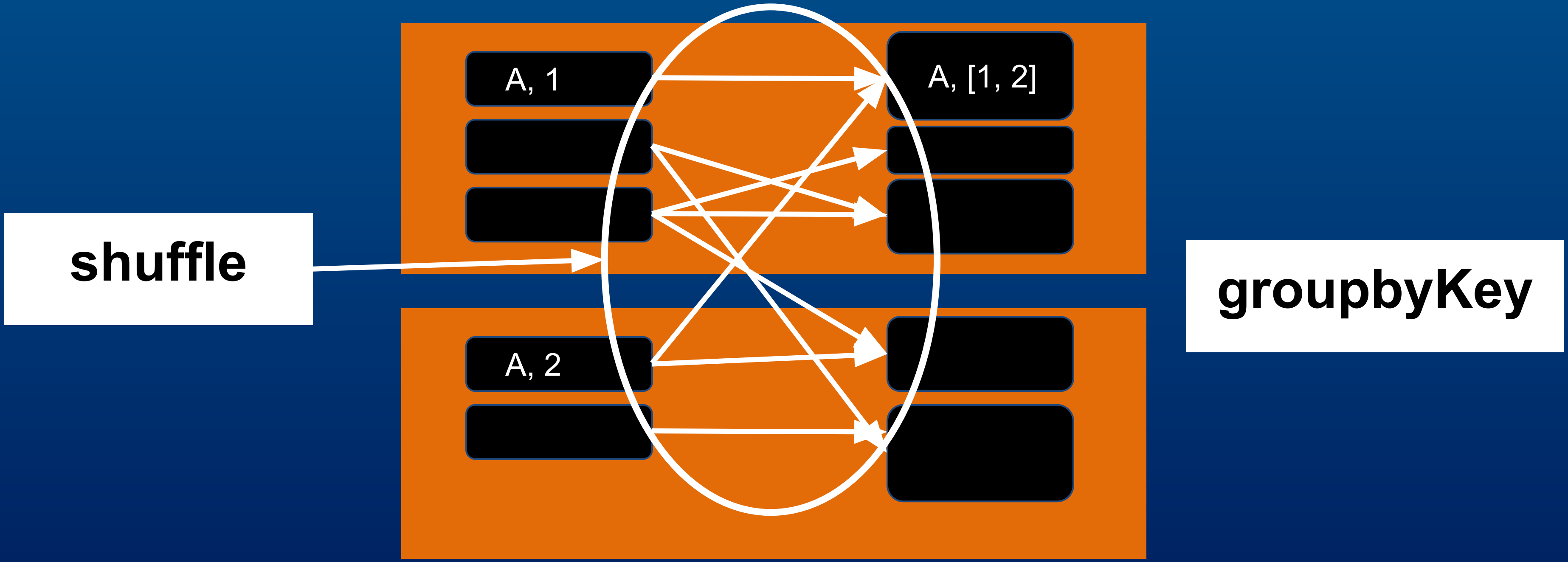
<MORE output>

```
for k,v in pairs_RDD.groupByKey().collect():
    print "Key:", k, ",Values:", list(v)
```

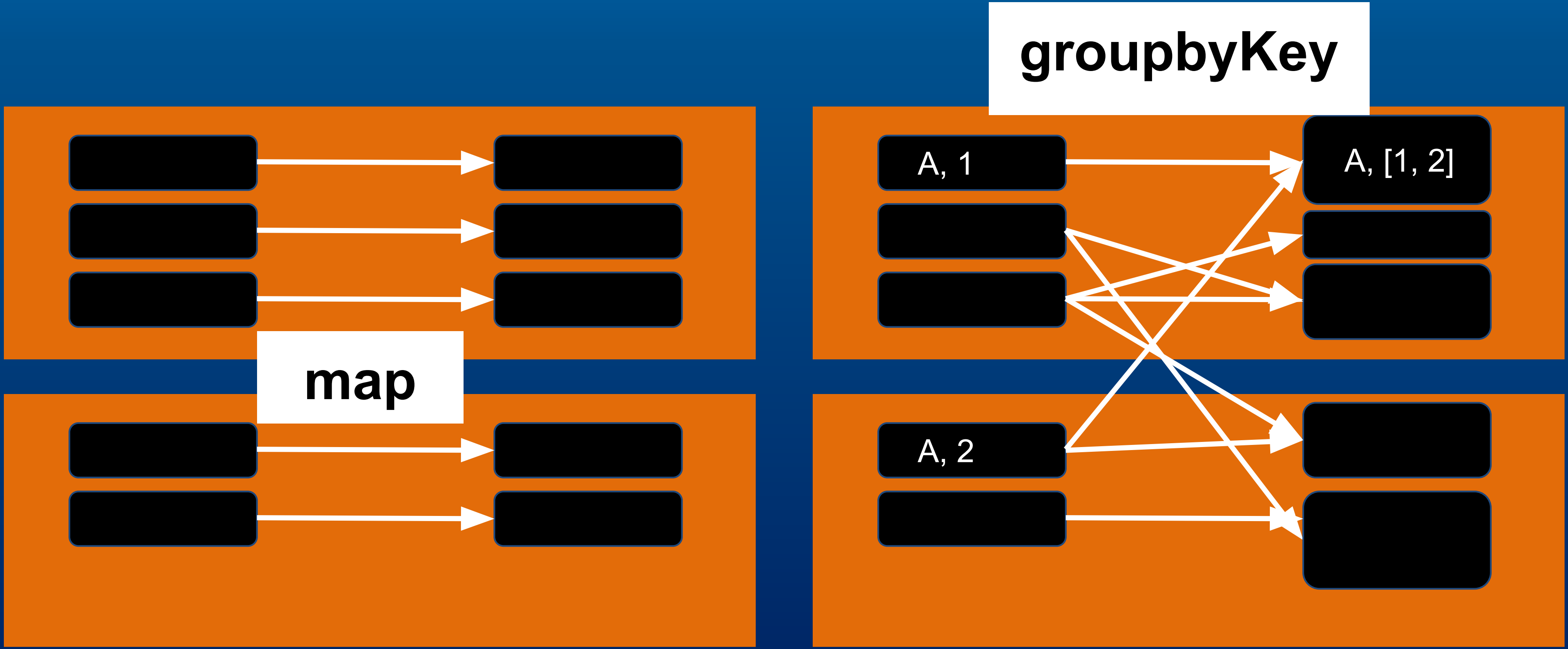
```
Out[]: Key: A , Values: [1]
Key: ago , Values: [1]
Key: far , Values: [1, 1]
Key: away , Values: [1]
Key: in , Values: [1]
Key: long , Values: [1]
Key: a , Values: [1]
<MORE output>
```

groupByKey

groupByKey : (K, V) pairs => (K, iterable of all V)



Narrow vs Wide



Wide transformations

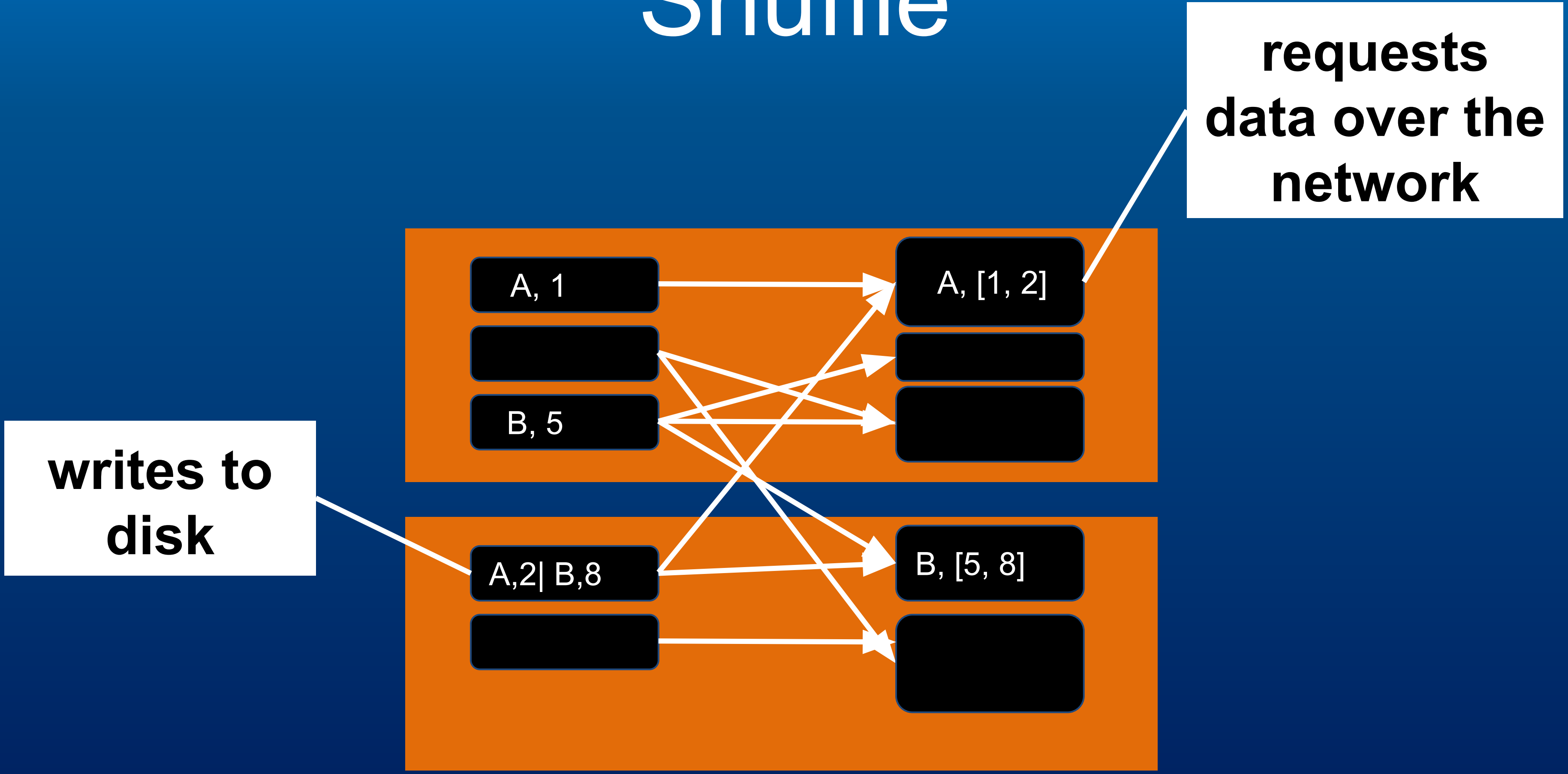
- groupByKey : (K, V) pairs => (K, iterable of all V)
- reduceByKey(func) : (K, V) pairs => (K, result of reduction by func on all V)
- repartition(numPartitions) : similar to coalesce, shuffles all data to increase or decrease number of partitions to numPartitions

Shuffle

Shuffle

- Global redistribution of data
- High impact on performance

Shuffle


requests data over the network
writes to disk
A, 1
B, 5
A,2| B,8
A, [1, 2]
B, [5, 8]

Know shuffle, avoid it

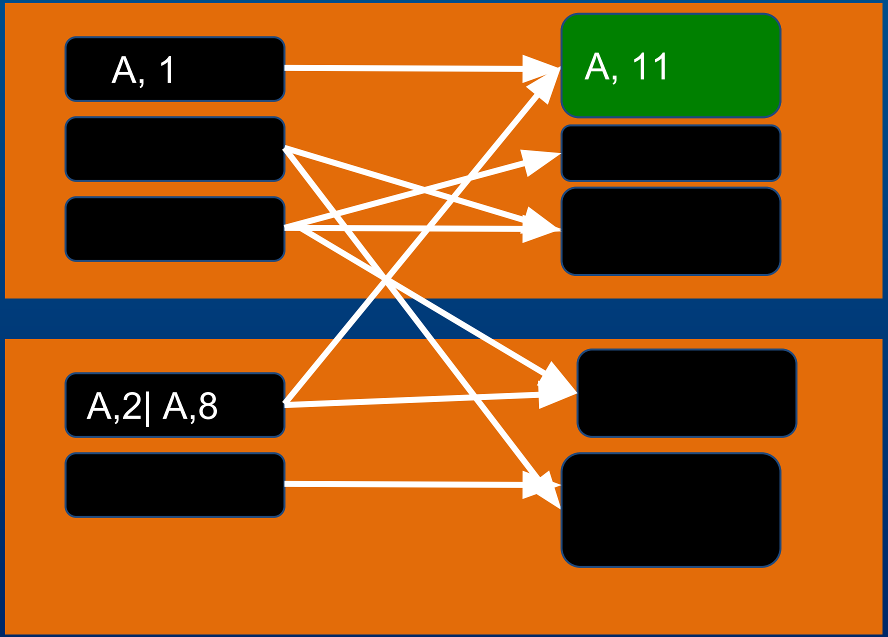
- Which operations cause it?
- Is it necessary?

Really need groupByKey?

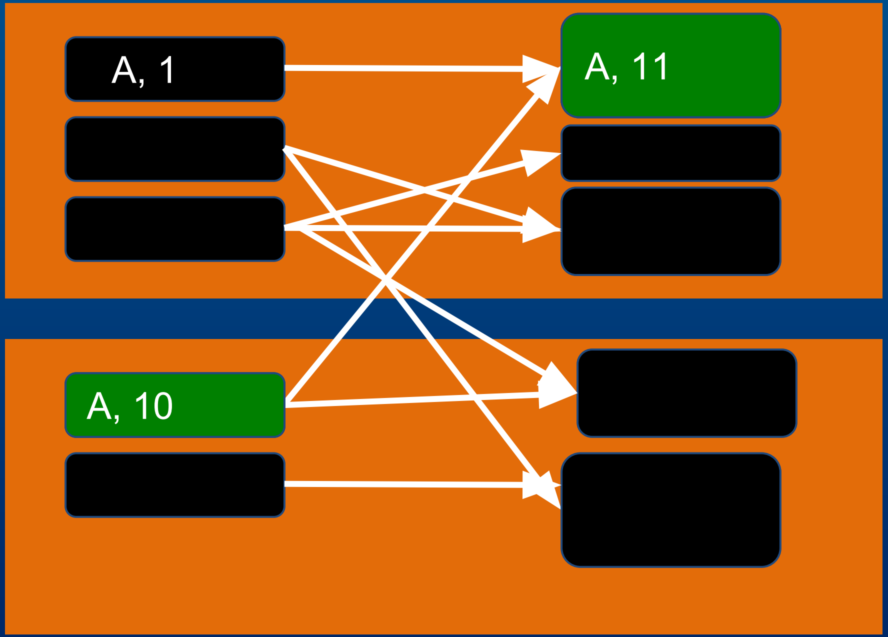
groupByKey: (K, V) pairs => (K, iterable of all V)

if you plan to call reduce later in the pipeline,
use reduceByKey instead.

groupByKey + reduce



reduceByKey



Million Songs Dataset

- Metadata about 1 million songs
- Fields like title, artist, year, loudness, hotness
- Created by Columbia University
- Dedicated to Machine Learning studies
- 280 GB compressed

Example Input

input file: “songs.tsv”

0	.4
1	.9
2	.2
3	.12
4	.55
5	.98

Example (Serial)

```
lines = open("songs.tsv").readlines()

def extract_hotness(line):
        return float(line.split()[1])

songs_hotness =map(extract_hotness, lines)
max_hotness = max(songs_hotness)
```

Example (Lambda)

```
def extract_hotness(line):
	return float(line.split()[1])

songs_hotness =map(extract_hotness, lines)

songs_hotness =map(lambda x:float(x.split()
[1]), lines)
```

Example (PySpark)

```
data = sc.textFile('songs.tsv')

def extract_hotness(line):
        return float(line.split()[1])

songs_hotness = data.map(extract_hotness)
max_hotness = songs_hotness.max()
```

Hands-on

Print all the values of hotness larger than .5

Example (PySpark)

```
songs_hotness.filter(lambda x:x>.5).collect()
[0.9, 0.55, 0.98]
```

Extract data from RDD

- collect() - copy all elements to the driver
- take(n) - copy first n elements
- saveAsTextFile(filename) - save to file
- reduce(func) - aggregate elements with func (takes 2 elements, returns 1)

Cache data in memory

```
max_hotness = songs_hotness.max()
min_hotness = songs_hotness.min()
```

Each operation reads the data back again, Spark does not keep intermediate results in memory. Can trigger cache with:

```
songs_hotness.cache()
```

Cached RDD

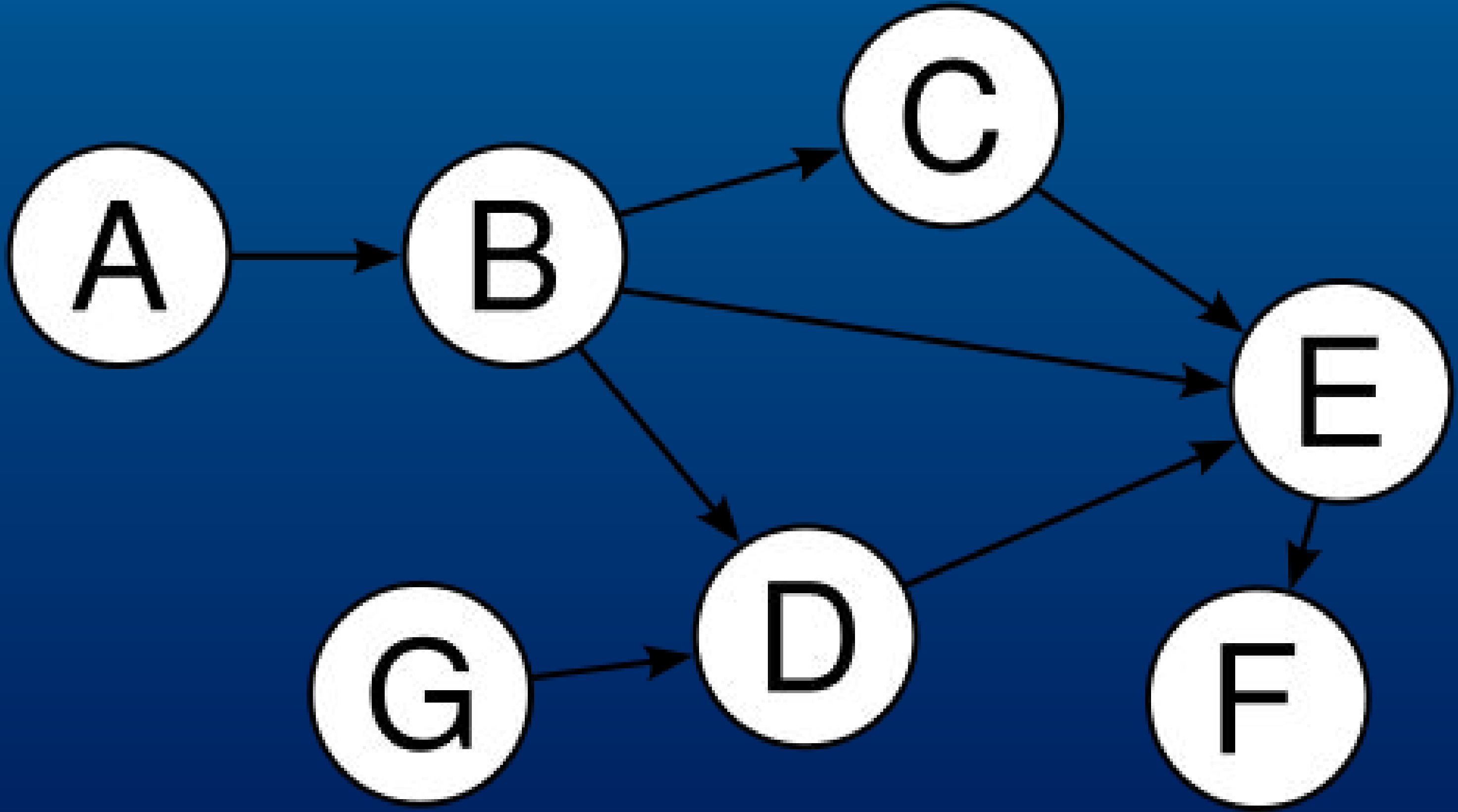
- Generally recommended after data cleaning
- Reusing cached data: 10x speedup
- Great for iterative algorithms
- If RDD too large, will only be partially cached in memory

```
data = sc.textFile('songs.tsv')

def extract_hotness(line):
        return float(line.split()[1])

songs_hotness = data.map(extract_hotness)
songs_hotness.cache()
max_hotness = songs_hotness.max()
```

Spark is lazy

Directed Acyclic Graph Scheduler

Directed Acyclic Graphs



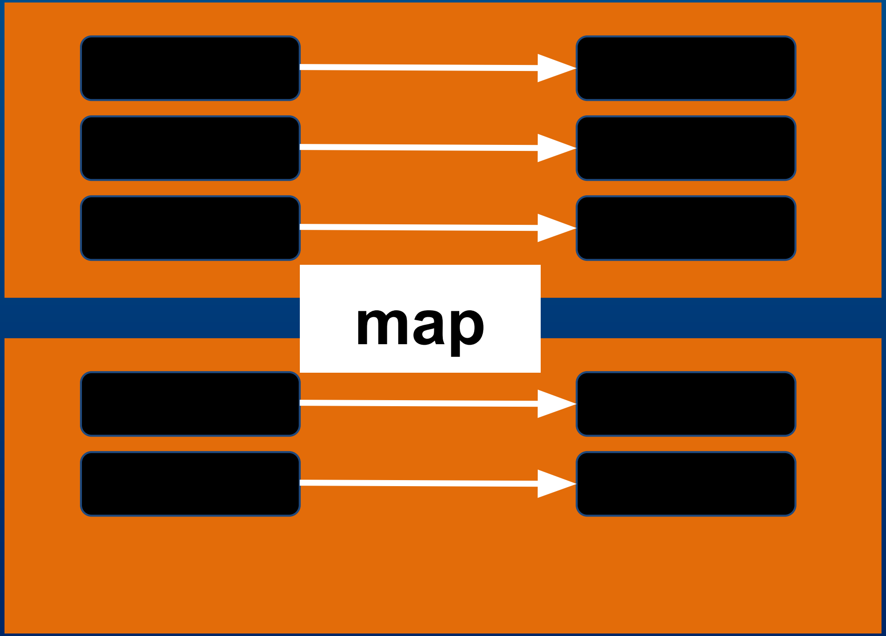
Directed Acyclic Graphs

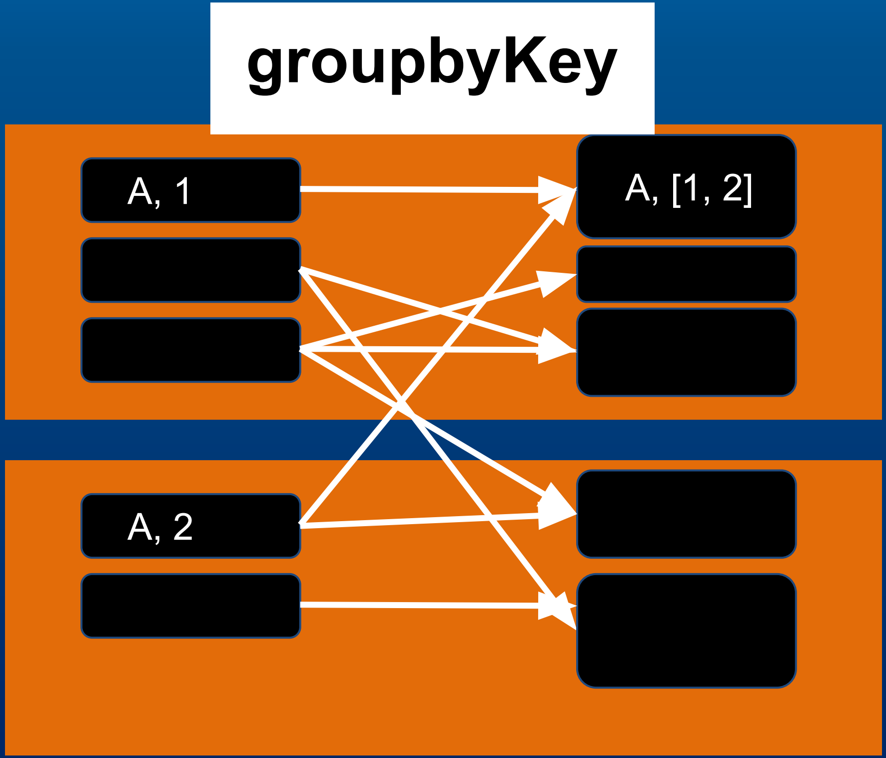
Track **dependencies**!
(also known as lineage or provenance)

DAG in Spark

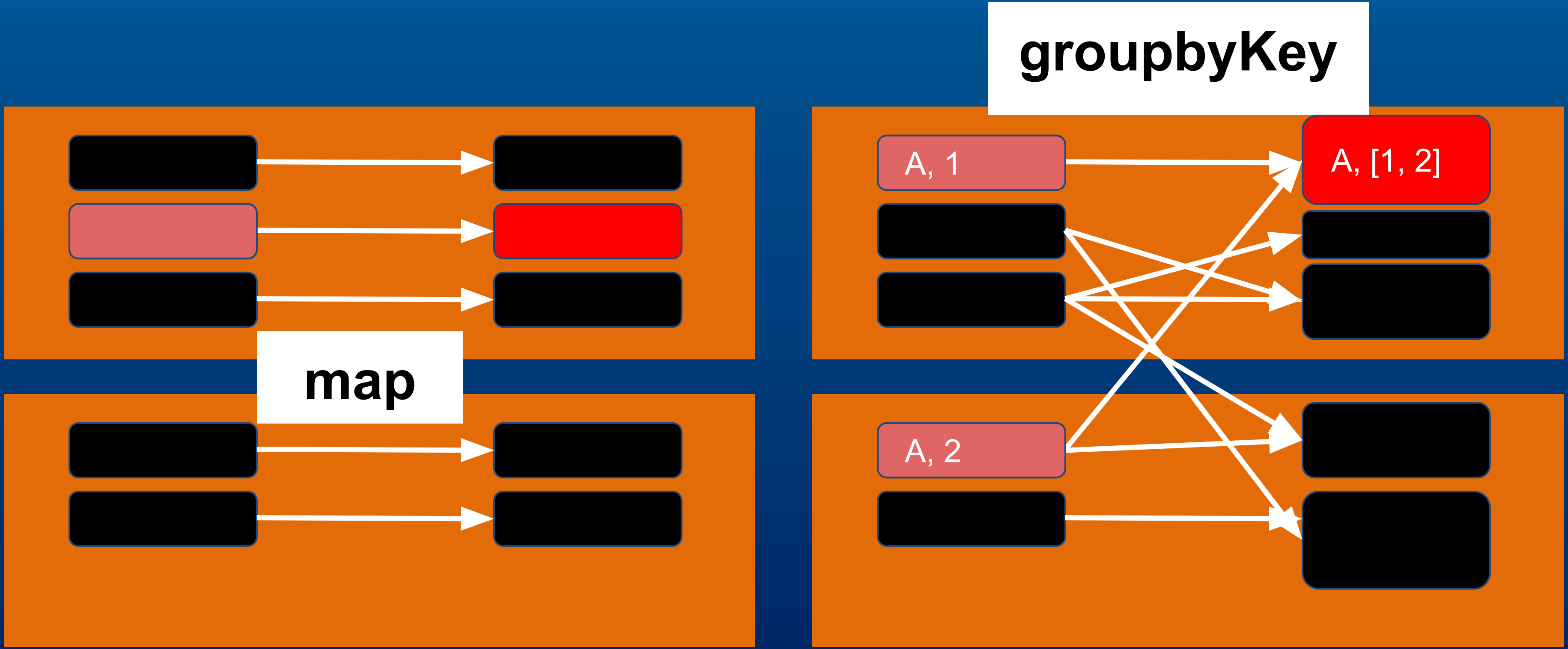
- nodes are RDDs
- arrows are Transformations

Narrow vs Wide





Narrow vs Wide



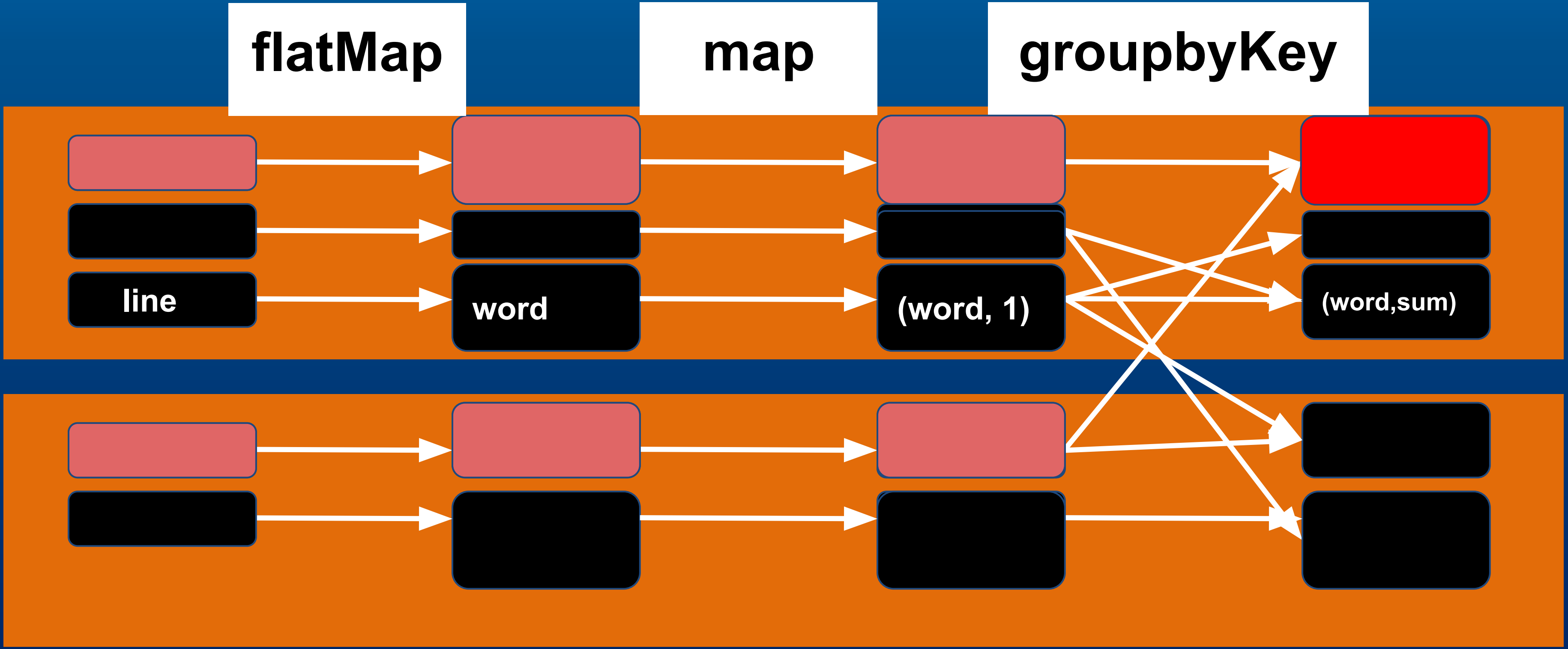
Transformations of (K,V) pairs

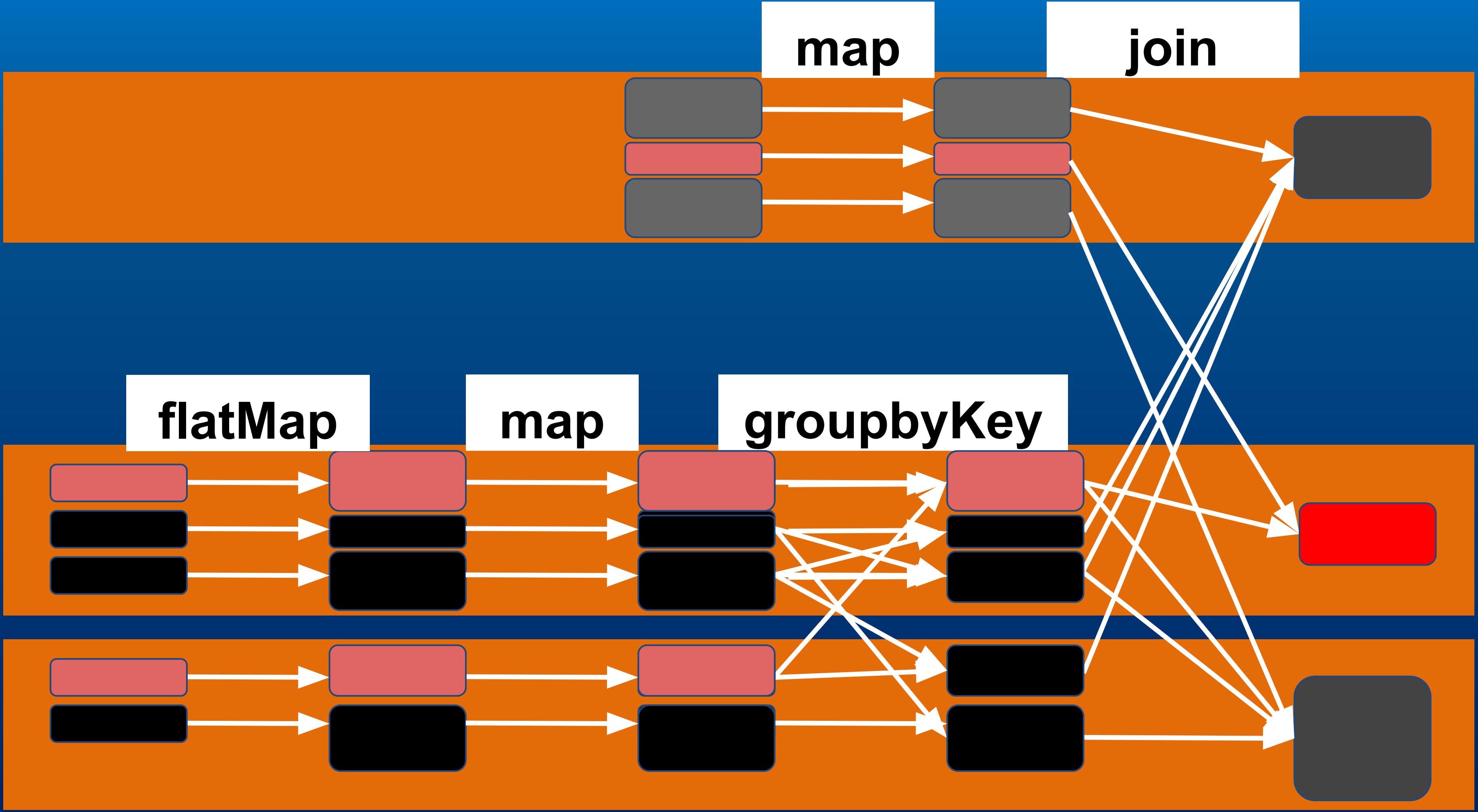
```
def create_pair(word):
    return (word, 1)

pairs_RDD=text_RDD.flatMap(split_words).map(create_pair)
```

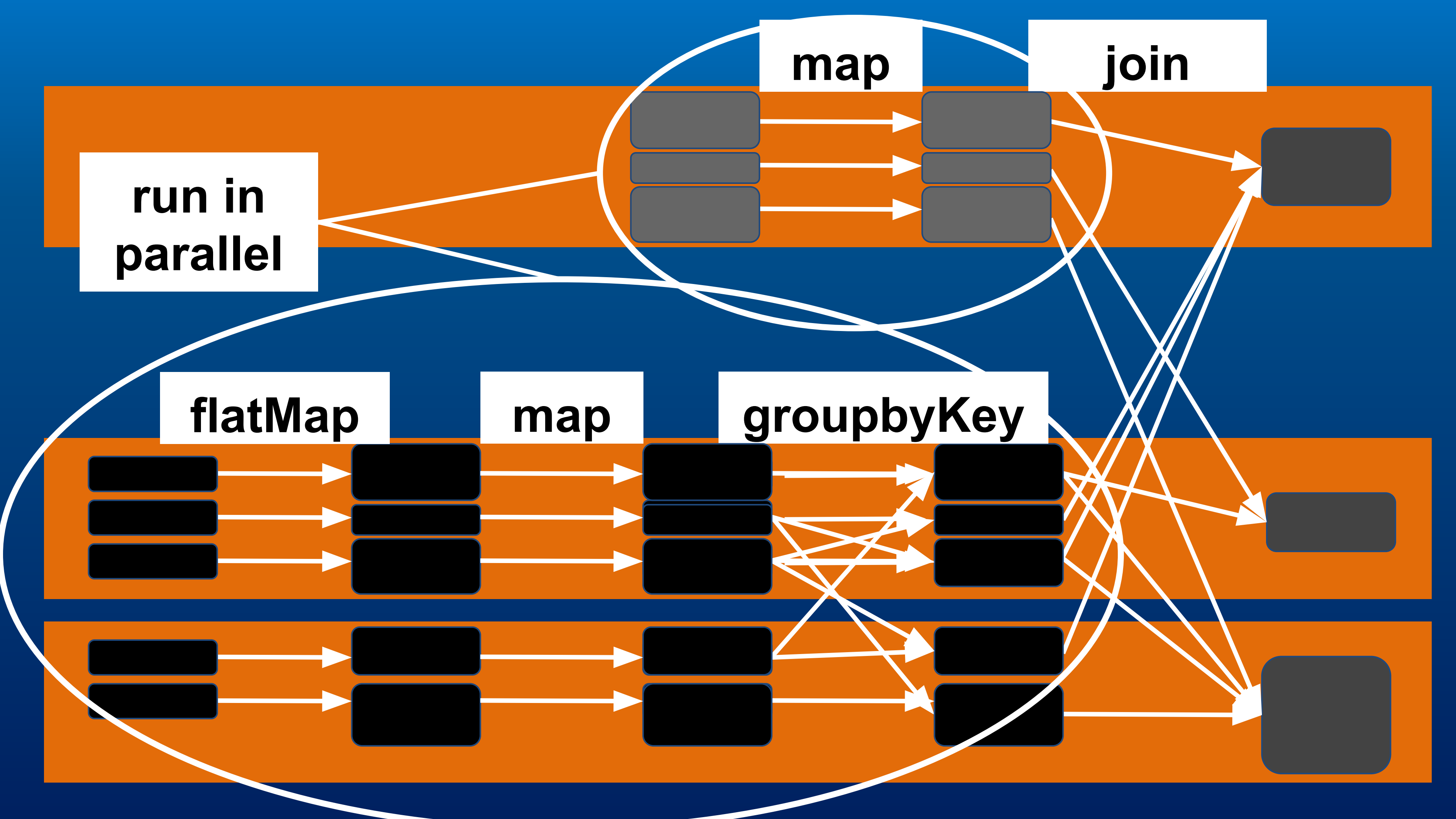
```
for k,v in pairs_RDD.groupByKey().collect():
    print "Key:", k, ",Values:", list(v)
```

```
Out[]: Key: A , Values: [1]
Key: ago , Values: [1]
Key: far , Values: [1, 1]
Key: away , Values: [1]
Key: in , Values: [1]
Key: long , Values: [1]
Key: a , Values: [1]
<MORE output>
```

Spark DAG of transformations




map
join
flatMap
map
groupbyKey


map
join
run in parallel
flatMap
map
groupbyKey

Broadcast variables

- Large variable used in all nodes, possibly in many functions
- Transfer just once per Executor
- For example large configuration dictionary or lookup table

```
config = sc.broadcast({"order":3, "filter":True})
config.value
```

Accumulators

- Common pattern of accumulating to a variable across the cluster

```
accum = sc.accumulator(0)
sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
accum.value
10
```

SparkUI

from your local machine:

$ ssh username@comet.sdsc.edu -L 4040:IP:
4040

Open browser at localhost:4040

Spark SQL

Tabular data processing in Spark

What is Spark SQL?

- High-level interface for structured data (i.e. tables)
- Provides Dataframes (data organized in columns), ~pandas, R
- Runs distributed SQL queries

Advantages of Spark SQL

- Same interface in Java/Scala/Python/R
- Native speed even in Python/R
- More expressive code -> easier to maintain

Spark SQL demo

Open pyspark

Hands-on

1. find maximum hotness for each decade
2. find how many songs for each decade

Spark MLlib

Machine Learning with Spark

Spark MLlib introduction

- Machine learning library
- Built on top of Spark
- Distributed linear algebra primitives:
    - Labeled points [y, X]
    - Dense vectors and matrices
    - Sparse vectors, matrices, block matrices

Spark MLlib features

- linear SVM and logistic regression
- classification and regression tree
- random forest and gradient-boosted trees
- recommendation via alternating least squares
- clustering via k-means, Gaussian mixtures, and power iteration clustering
- singular value decomposition
- linear regression with L1- and L2-regularization

Spark MLlib demo

- open pyspark

Hands-on

- print the KMeans cluster centers (k=4) adding also the year as a features column

Thanks

Questions?

Andrea Zonca
zonca@sdsc.edu

View partitioning

- glom gathers all data in a partition as a list:

```
songs_hotness.repartition(2).glom().collect()
[[0.4, 0.12, 0.55], [0.98, 0.9, 0.2]]
songs_hotness.repartition(3).glom().collect()
[[0.12], [0.55, 0.4], [0.9, 0.2, 0.98]]
```