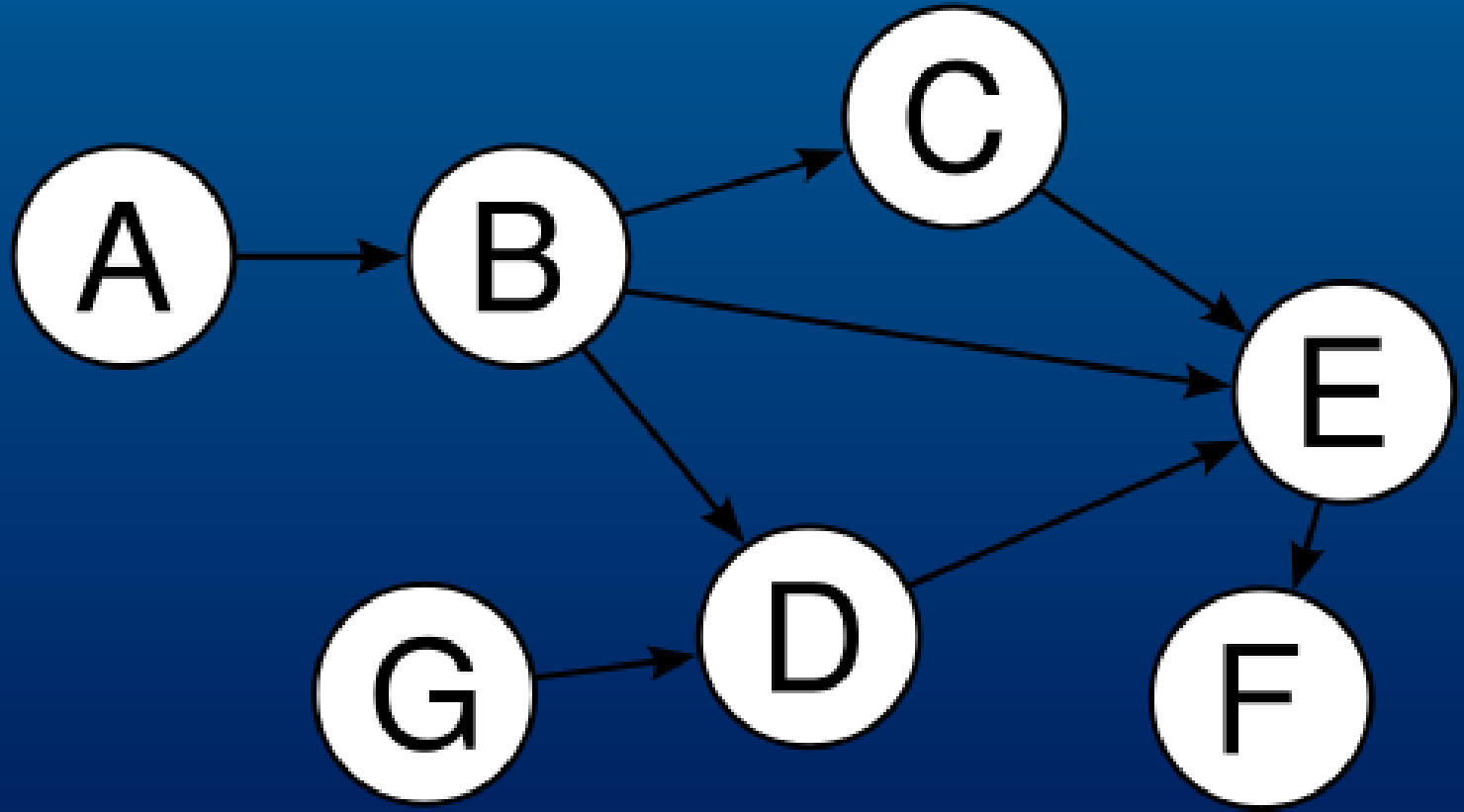
Directed Acyclic Graph Scheduler

Directed Acyclic Graphs



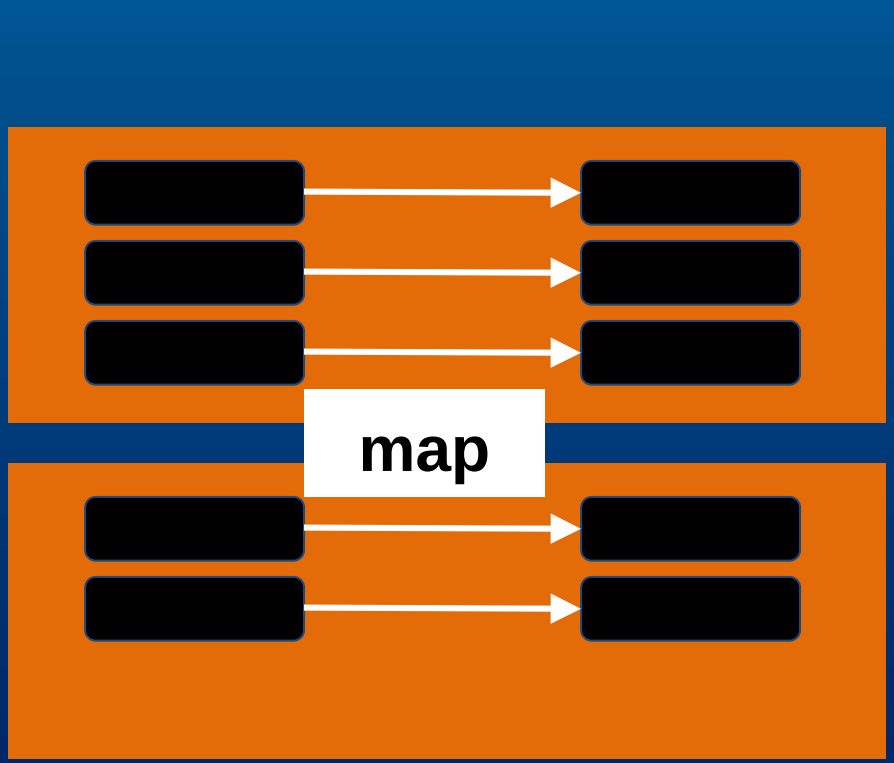
Directed Acyclic Graphs

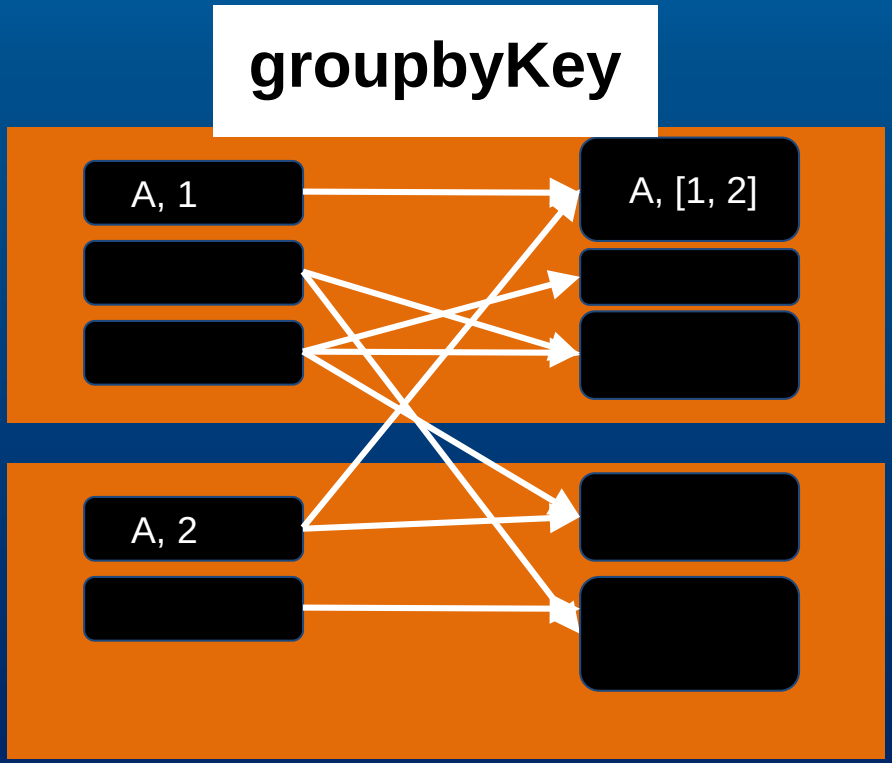
Track **dependencies**!
(also known as lineage or provenance)

DAG in Spark

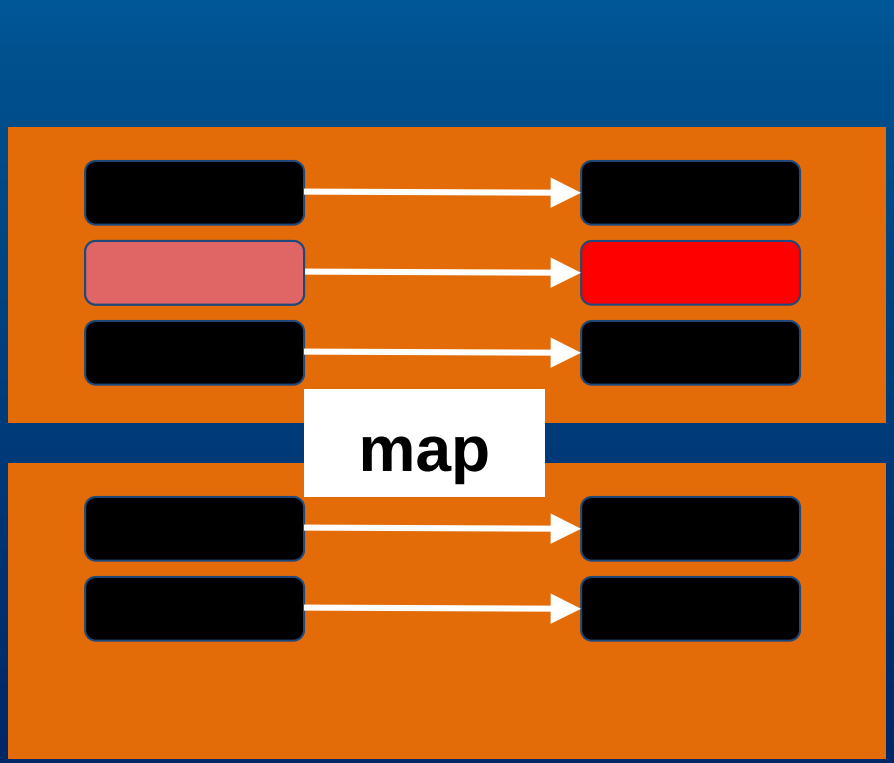
- nodes are RDDs
- arrows are Transformations

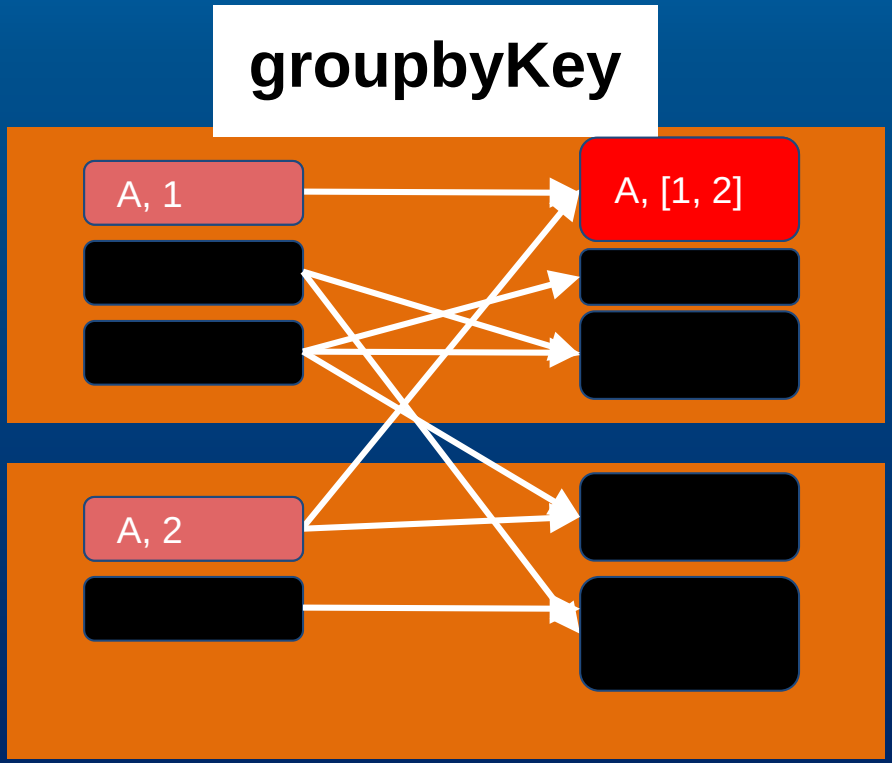
Narrow vs Wide





Narrow vs Wide





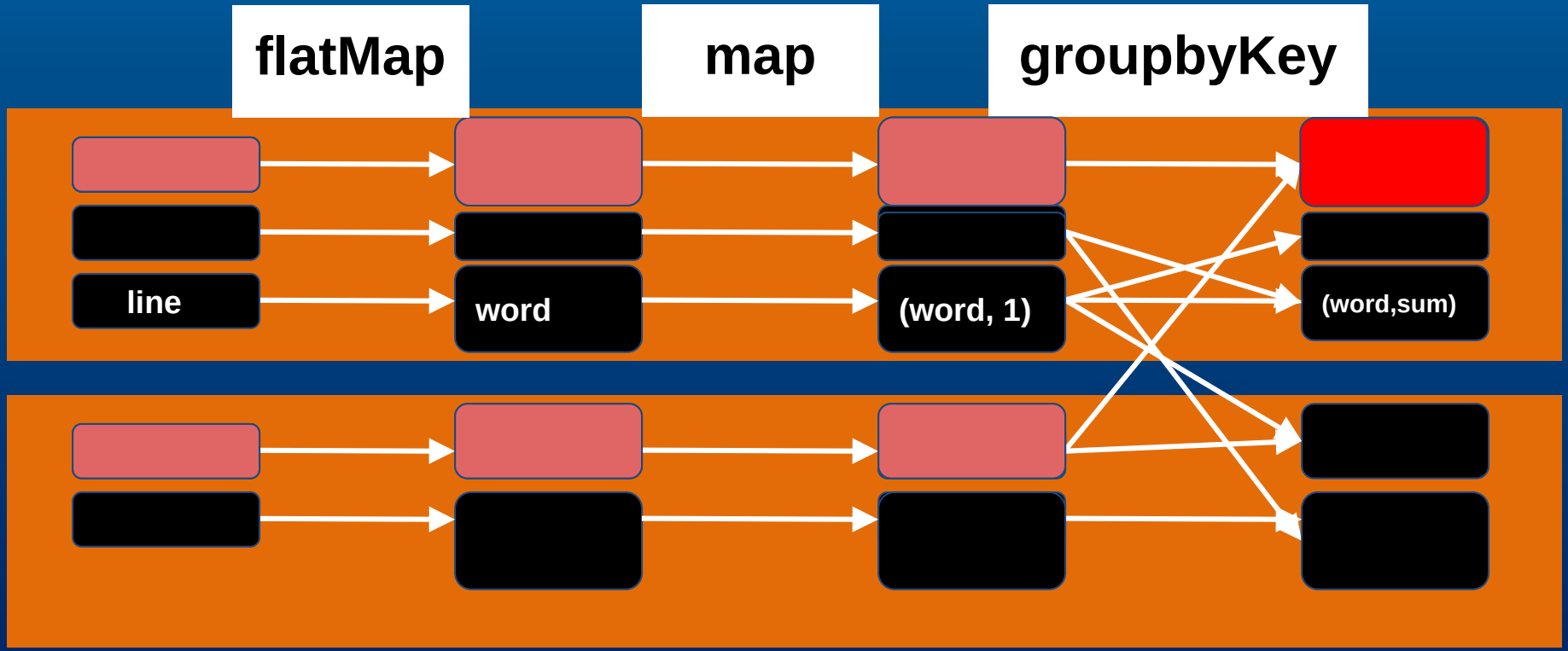
Transformations of (K,V) pairs

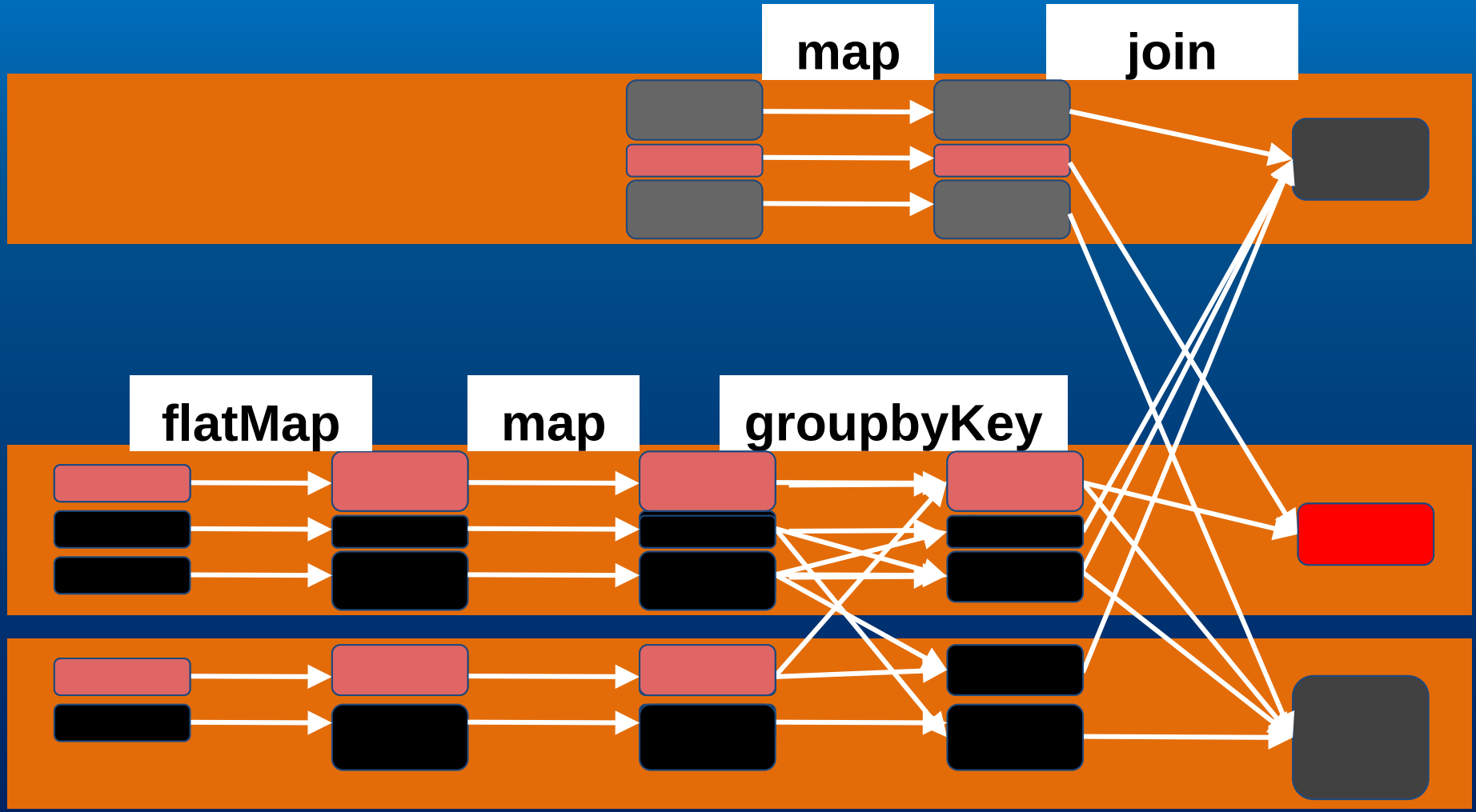
```
def create_pair(word):
    return (word, 1)

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

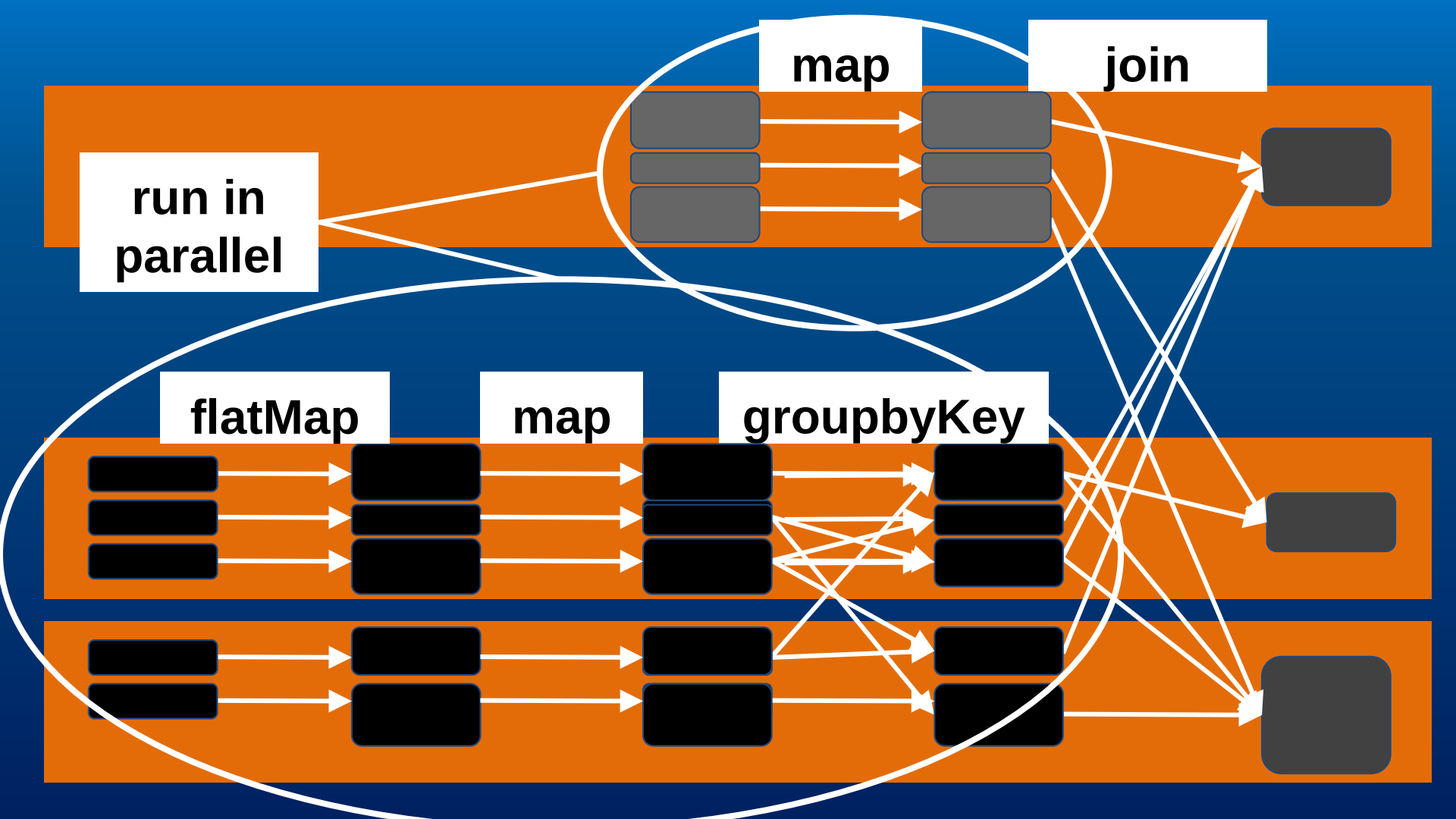
```
for k,v in pairs_RDD.groupByKey().collect():
    print "Key:", k, ",Values:", list(v)
```

```
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



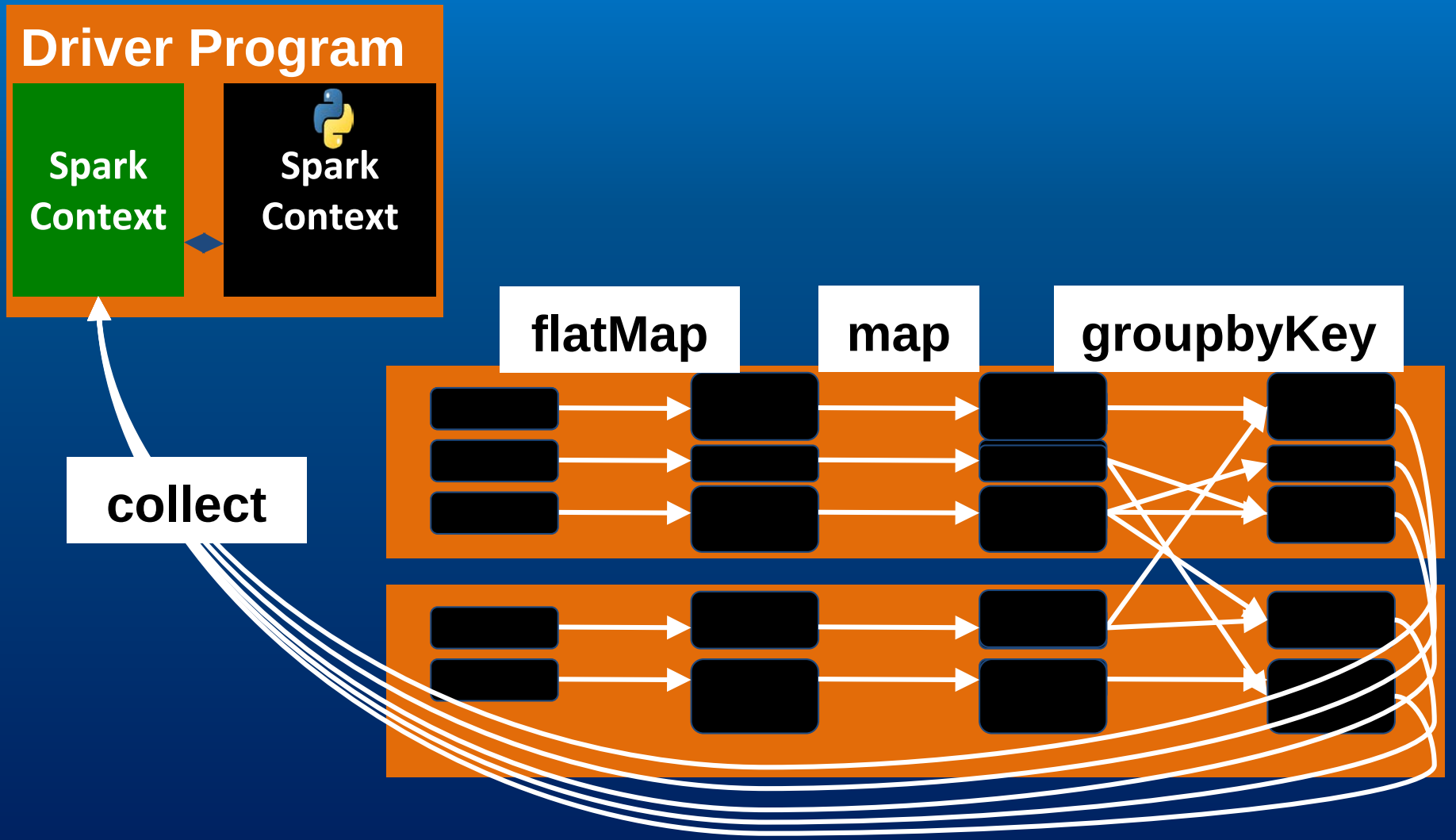

map
join
flatMap
map
groupbyKey


map
join
run in
parallel
flatMap
map
groupbyKey

Actions

What is an action

- Final stage of workflow
- Triggers execution of the DAG
- Returns results to the Driver or writes to HDFS


Driver Program
Spark Context
Spark Context
flatMap
map
groupbyKey
collect

Actions

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

Caching

Caching

- By default each job re-processes from HDFS
- Mark RDD with .cache()
- Lazy

When?

- Generally not the input data
- Do validation and cleaning
- Cache for iterative algorithm

How?

- Memory (most common)
- Disk (rare)
- Both (for heavy calculations)

Speedup

- Easily 10x or even 100x depending on application
- Caching is gradual
- Fault tolerant

Wordcount with caching

from HDFS:

```
text_RDD =
sc.textFile("/user/cloudera/input/testfile1")
```

```
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.cache()
```

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

wordcounts_RDD = pairs_RDD.reduceByKey(sum_counts)
```

First job:

```
wordcounts_RDD.collect()
```

Second job:

```
pairs_RDD.take(1)
```

Broadcast variables

Broadcast variables

- Large variable used in all nodes
- Transfer just once per Executor
- Efficient peer-to-peer transfer

Broadcast variable example

For example large configuration dictionary or lookup table:

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

Accumulators

Accumulator

- Common pattern of accumulating to a variable across the cluster
- Write-only on nodes

Accumulator example

```
accum = sc.accumulator(0)
def test_accum(x):
    accum.add(x)

sc.parallelize([1, 2, 3, 4]).foreach(test_accum)

accum.value
Out[]: 10
```