

Spark for Scientific Computing

A. Zonca - 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

Clone Github repository

- ssh to Comet
- git clone
<https://github.com/sdsc/sdsc-summer-institute-2016>

Interactive spark on Comet

```
$ cd ~/workshop/spark
```

submit a spark job with:

```
$ sbatch spark.cmd
```

check job status with

```
$ squeue -u $USER
```

Connect via browser

comet-XX-XX.sdsc.edu:8888

House prices

see `house_price.ipynb`

Resilient Distributed Dataset

Dataset

Data storage created from:
HDFS, S3, HBase, JSON, text,
Local hierarchy of folders

Or created transforming
another RDD

Resilient Distributed Dataset

Distributed

Distributed across the cluster
of machines

Divided in partitions, atomic
chunks of data

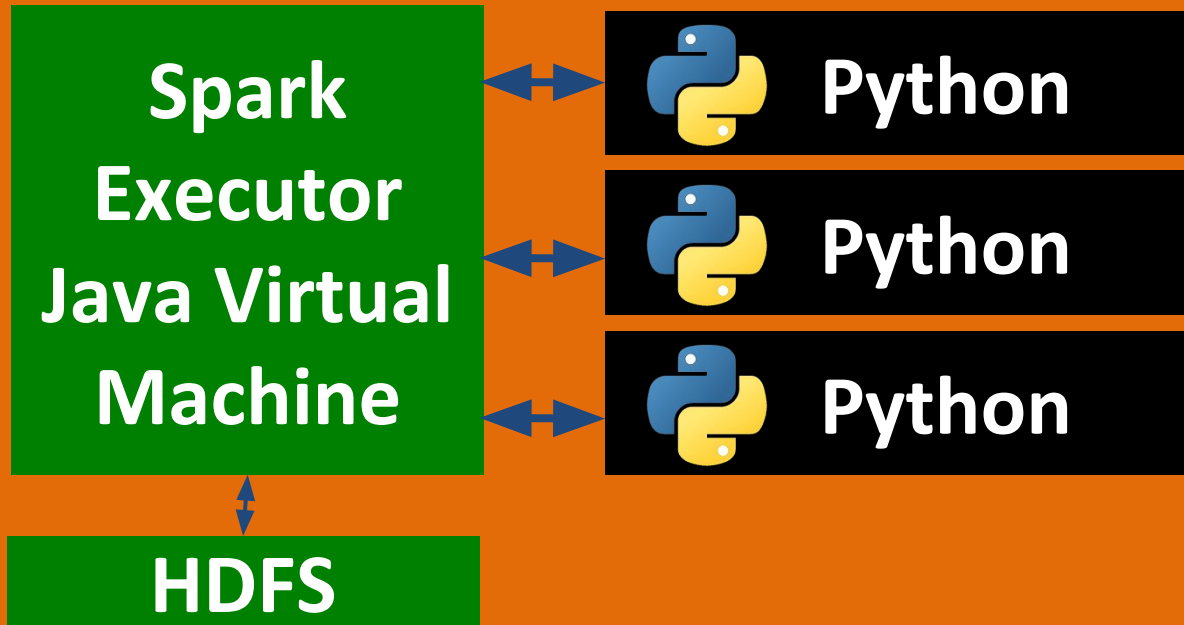
Resilient Distributed Dataset

Resilient

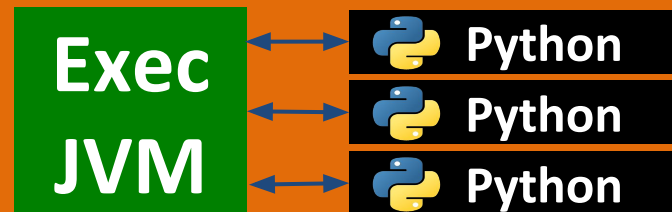
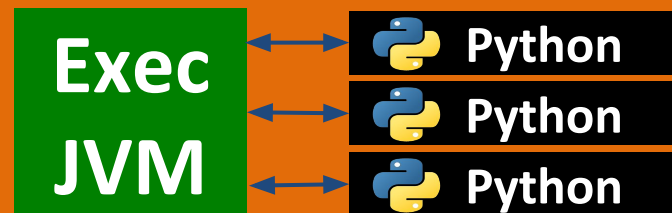
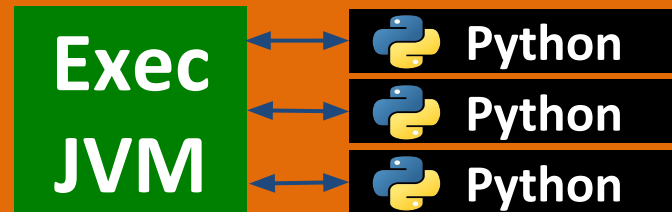
Recover from errors, e.g.
node failure, slow processes

Track history of each
partition, re-run

Worker Node






Worker Nodes






Cluster Manager
YARN/Standalone
Provision/Restart Workers

Worker Nodes




**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

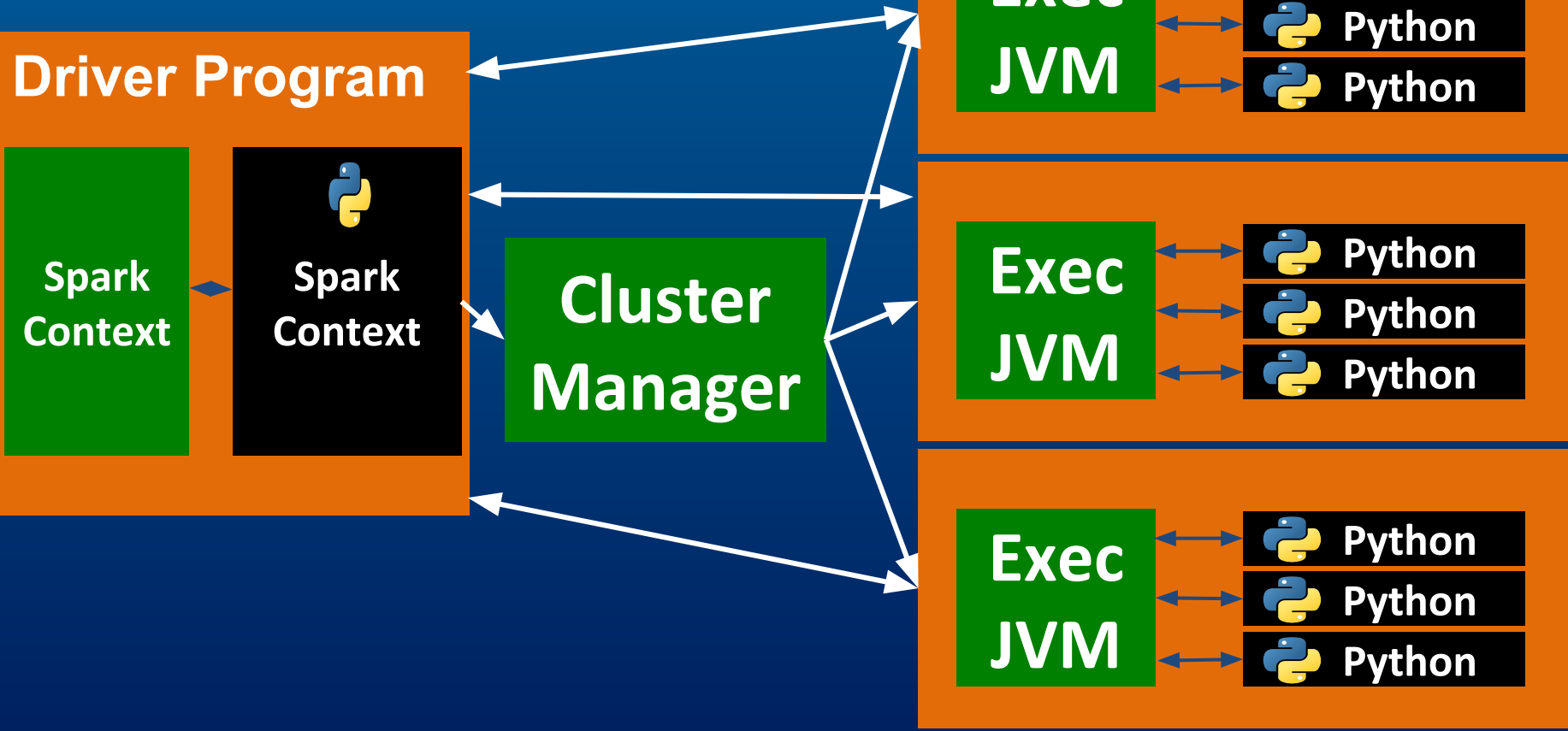
Python
Python
Python

Exec
JVM

Python
Python
Python

Exec
JVM


Python
Python
Python



Spark Local

Driver Program

Spark
Context


Spark
Context

Standalone

Exec
JVM

 Python

on Comet

Master node

Driver Program

Spark
Context

Spark
Context

Standalone
CM

Computing Nodes

Exec
JVM

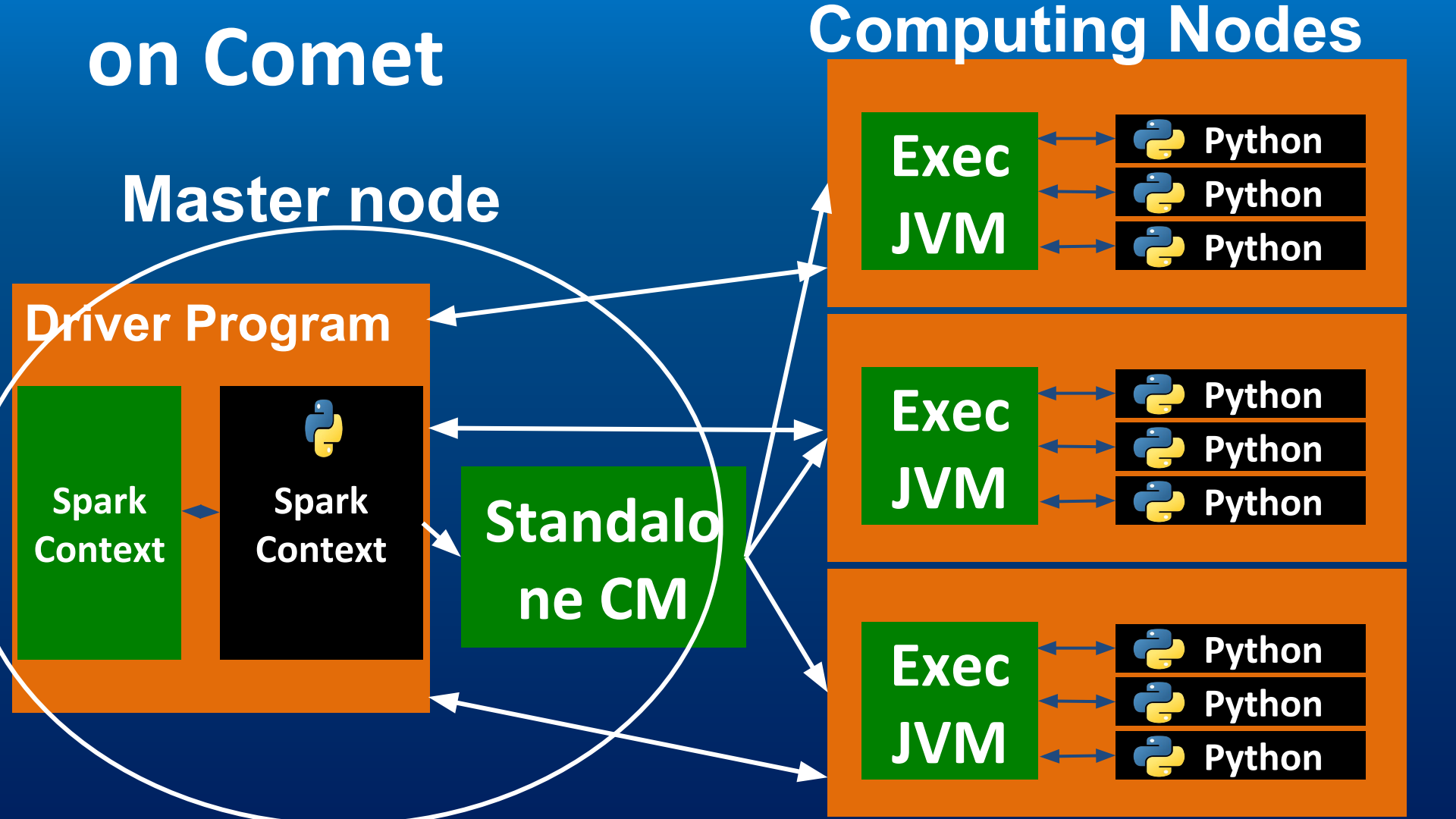
Python
Python
Python

Exec
JVM

Python
Python
Python

Exec
JVM

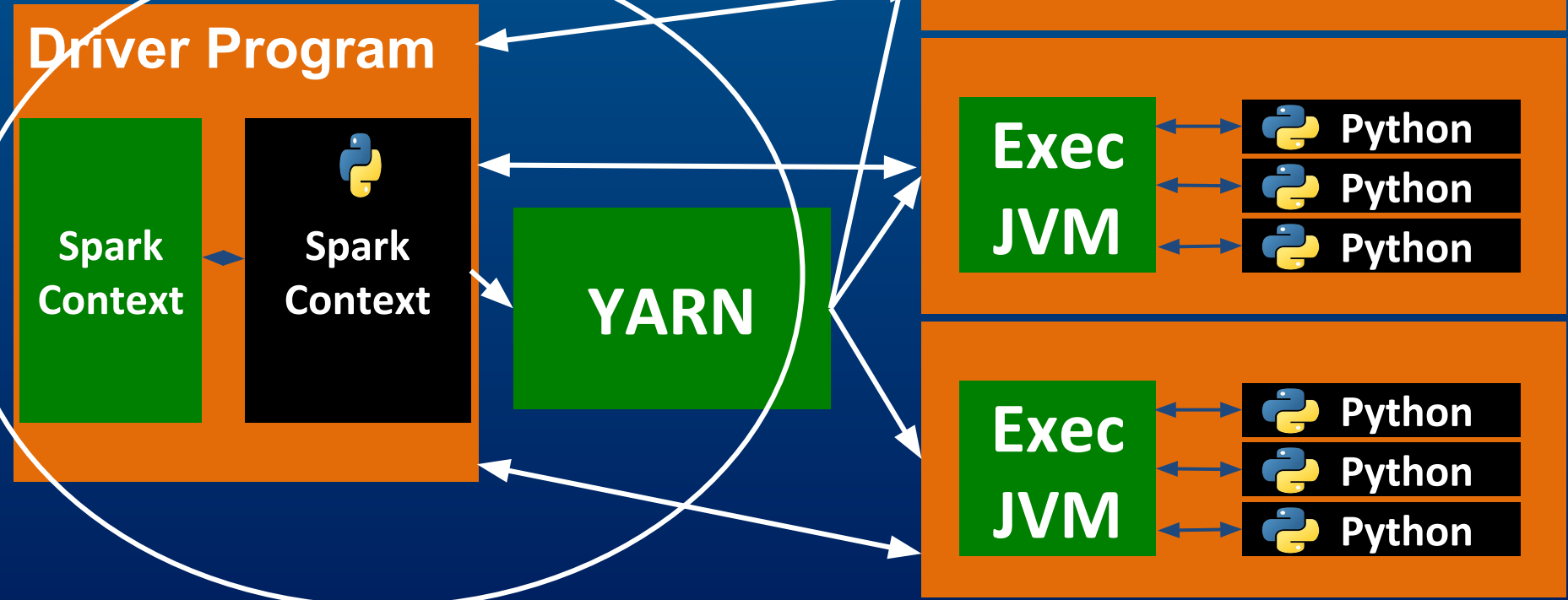
Python
Python
Python



on Amazon EMR

EC2 nodes

Master node

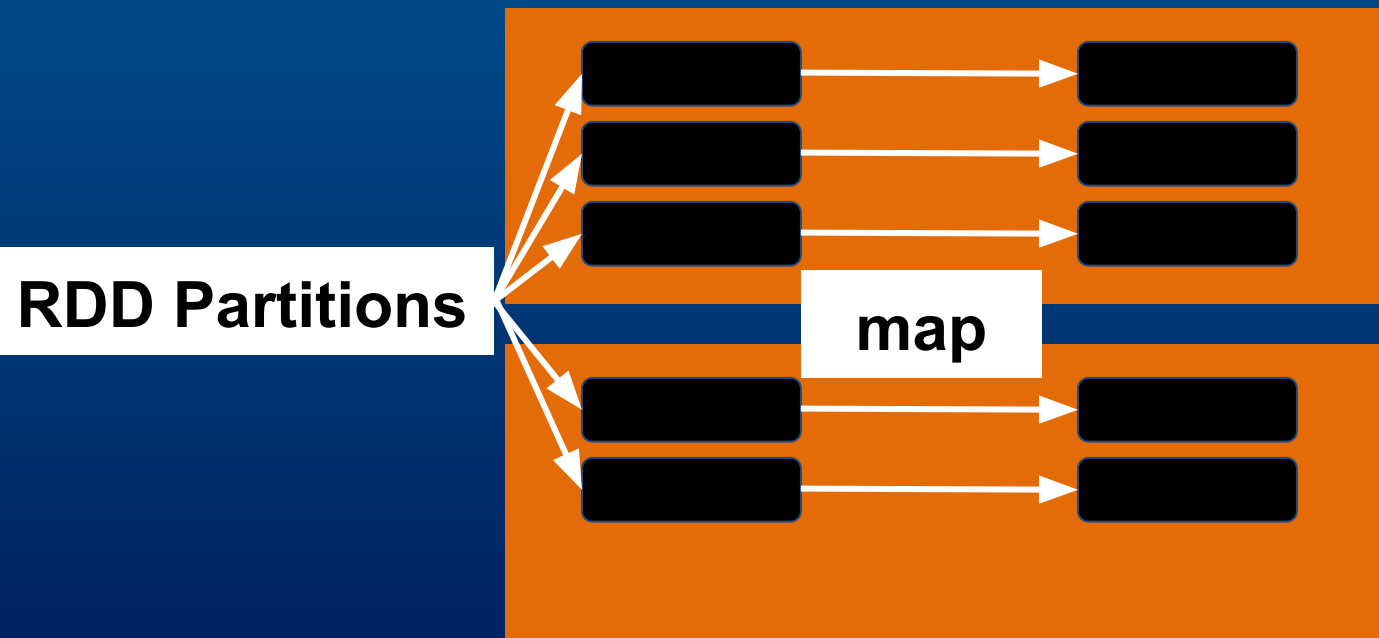


House price with HDFS

see `house_price_hdfs.ipynb`

map

map : apply function to each element of RDD

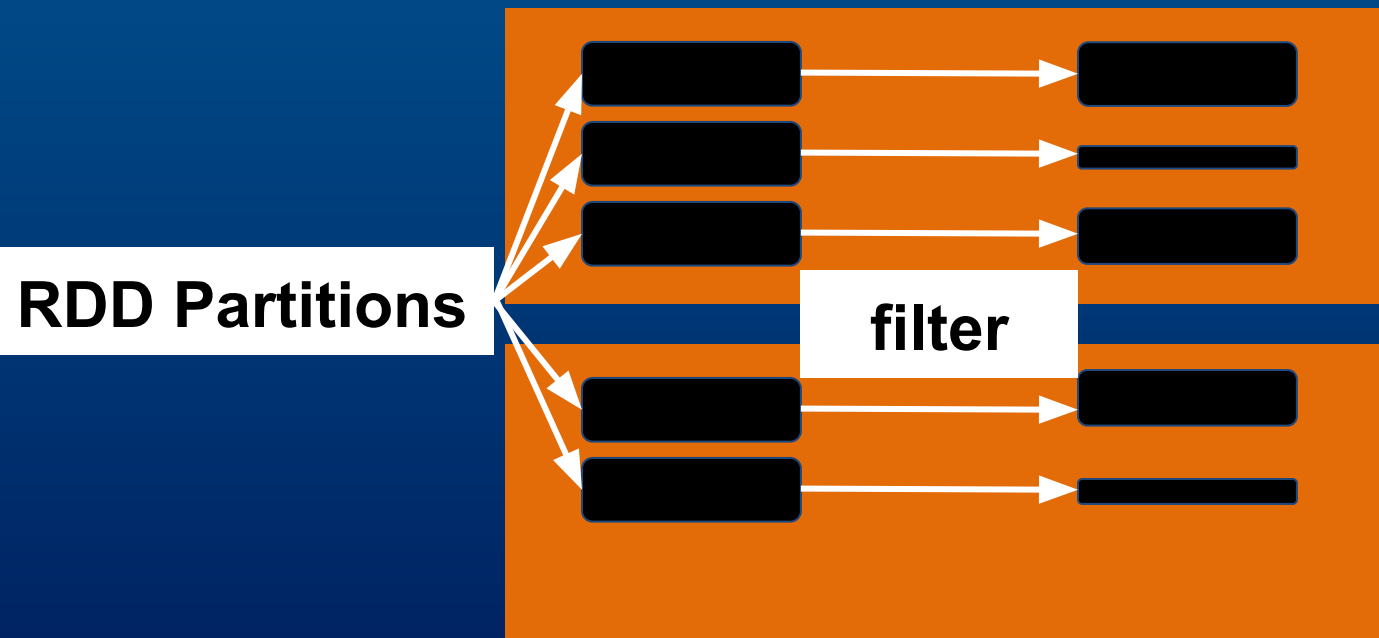


Other transformations

- `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

filter

filter : keep only elements where func is true



coalesce

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

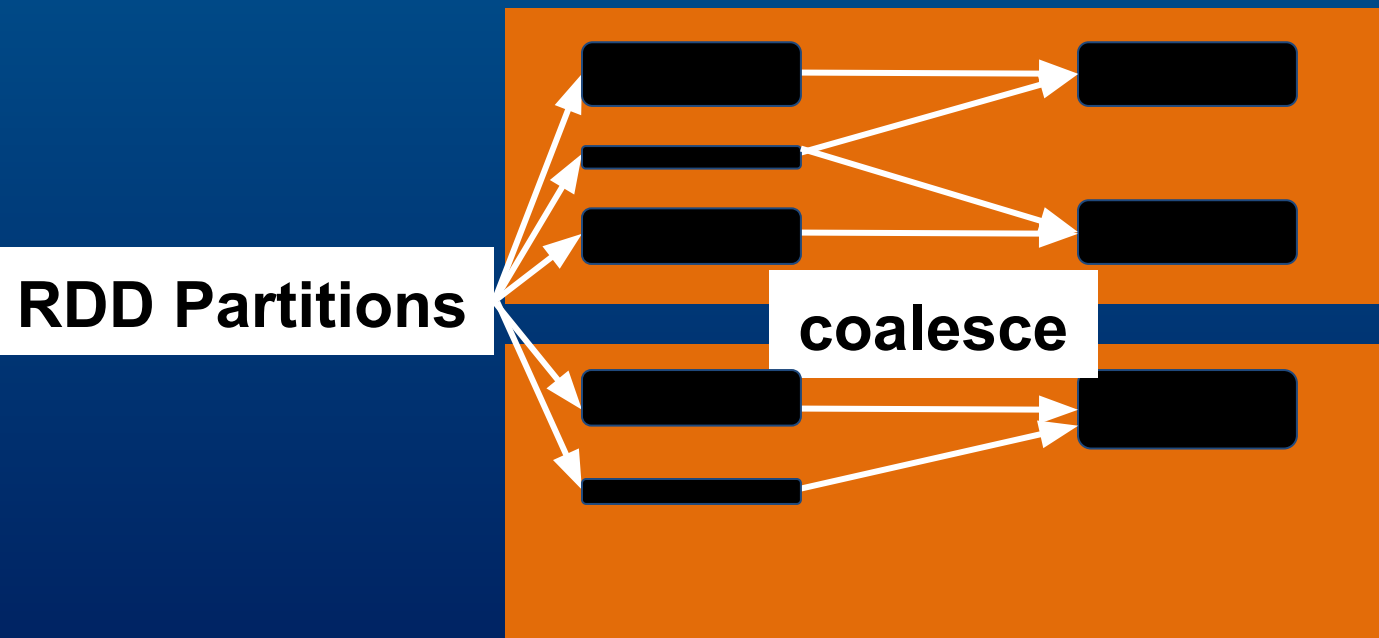
```
Out[]: [[0, 1], [2, 3], [4, 5], [6, 7, 8, 9]]
```

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

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

coalesce

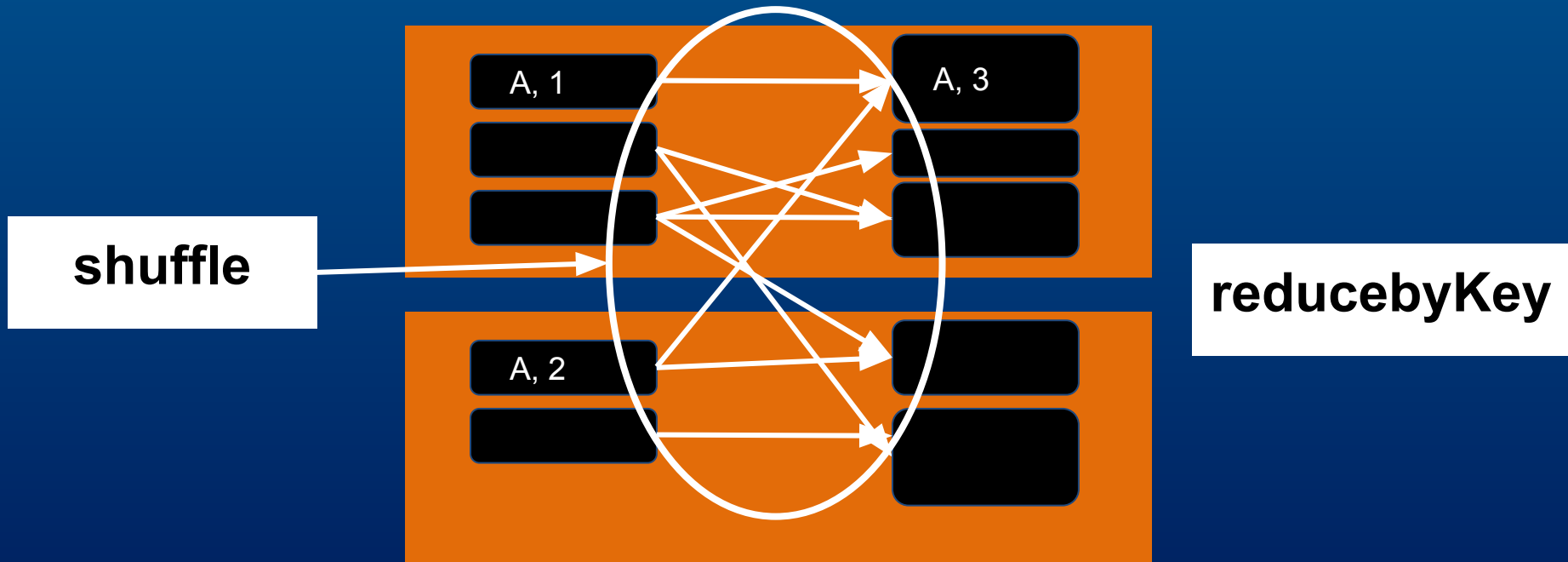
coalesce : reduce the number of partitions



Wide transformations

reduceByKey(func)

(K, V) pairs $\Rightarrow (K, \text{reduce } V \text{ with func})$



Narrow

vs

Wide

map

reduceByKey(sum)

A, 1

A, 3

A, 2

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

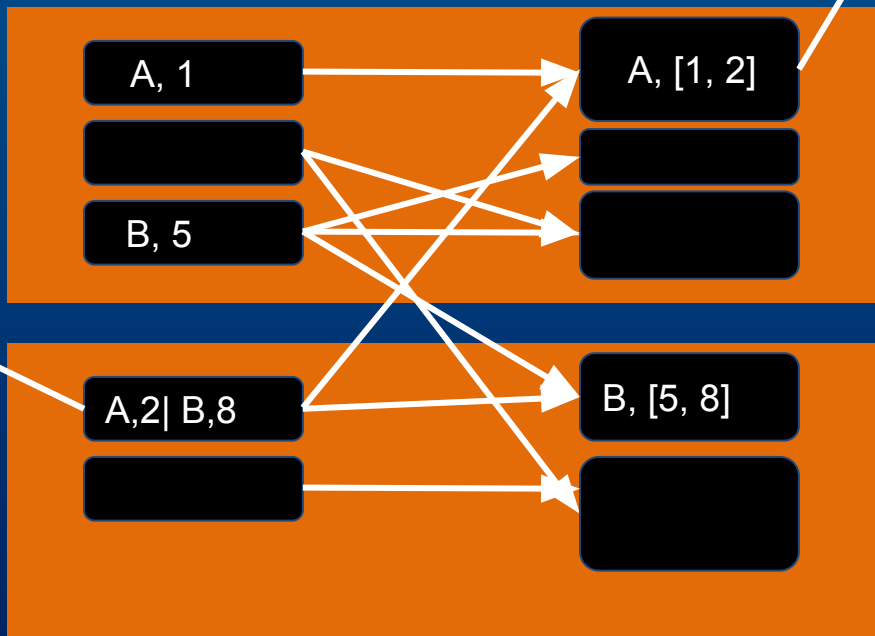
Shuffle

- . Global redistribution of data
- . High impact on performance

Shuffle

**requests
data over the
network**

**writes to
disk**



Know shuffle, avoid it

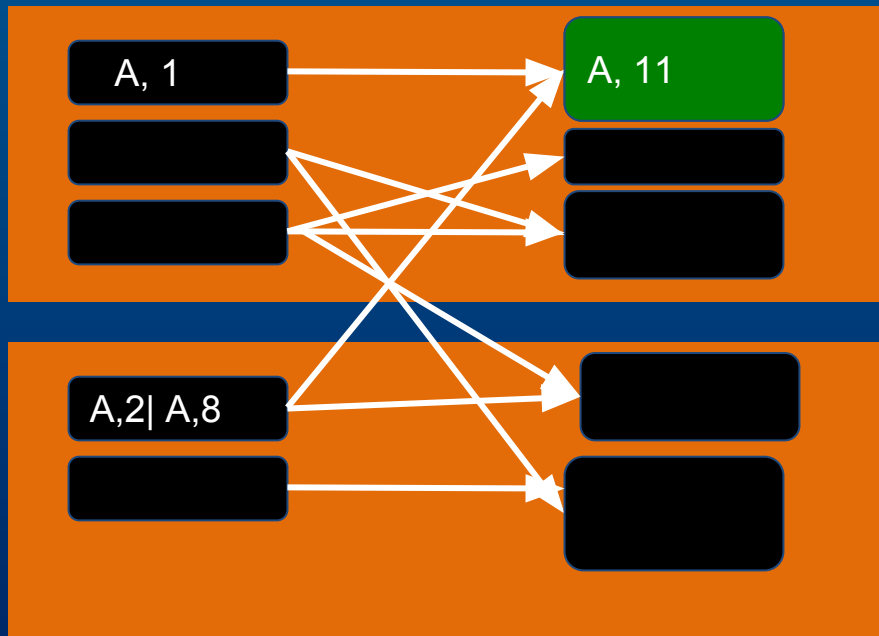
- . Which operations cause it?
- . Is it necessary?

Really need groupByKey?

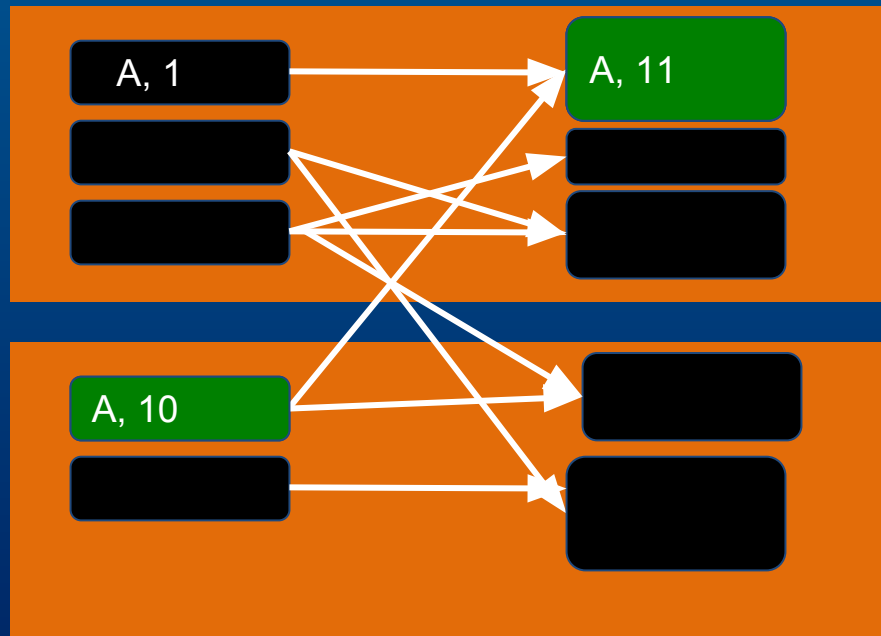
`groupByKey`: (K, V) pairs => (K, iterable of all V)

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

groupByKey + reduce



reduceByKey



Extract data from RDD

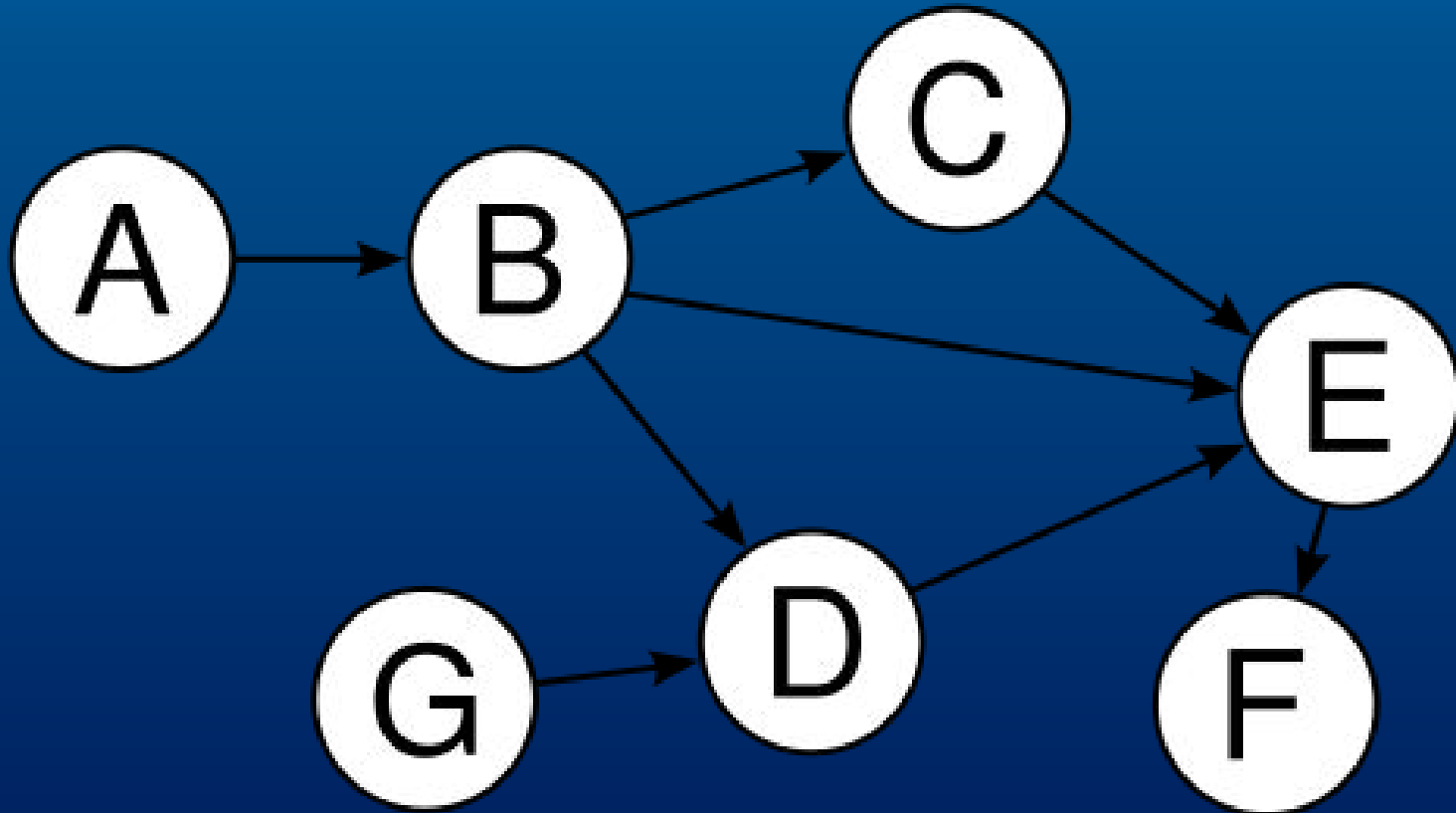
- `collect()` - copy all elements to the driver
- `take(n)` - copy first n elements
- `saveAsTextFile(filename)` - save to file
- `reduce(func)` - aggregate elements with func (takes 2 elements, returns 1)

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

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

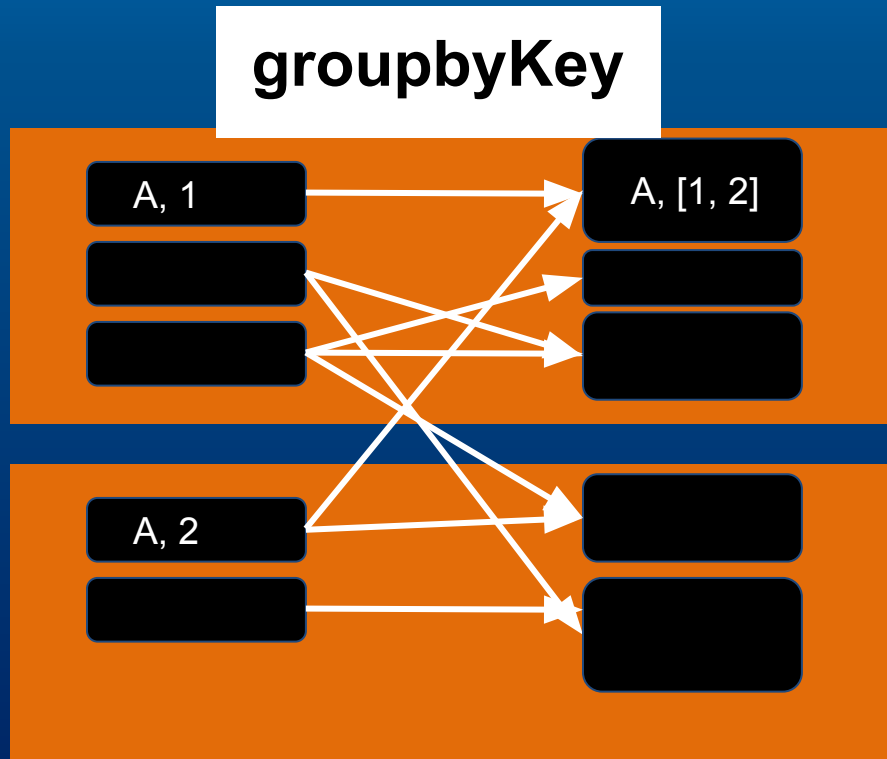
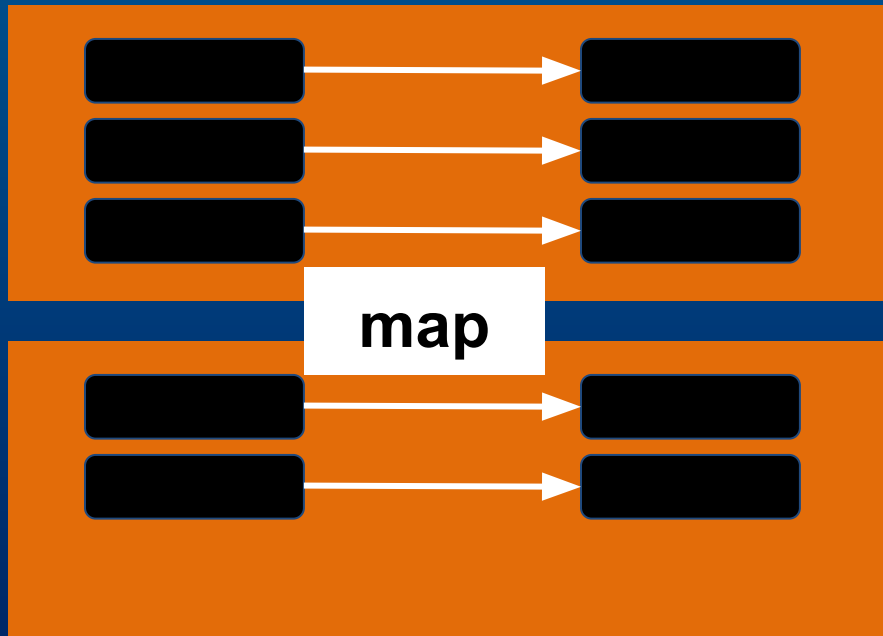
Narrow

vs

Wide

map

groupByKey



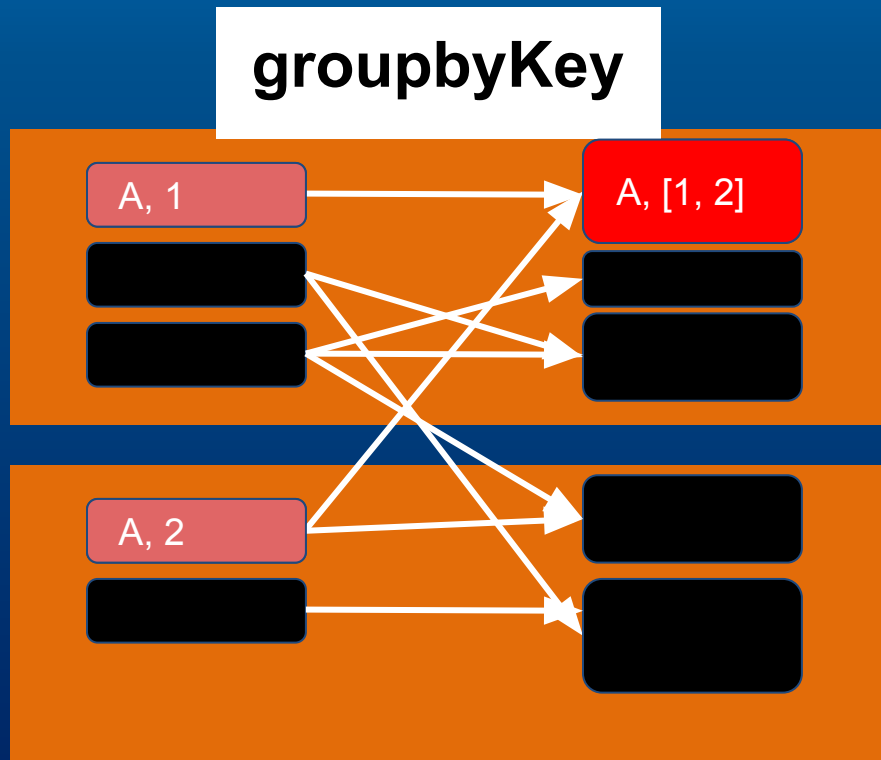
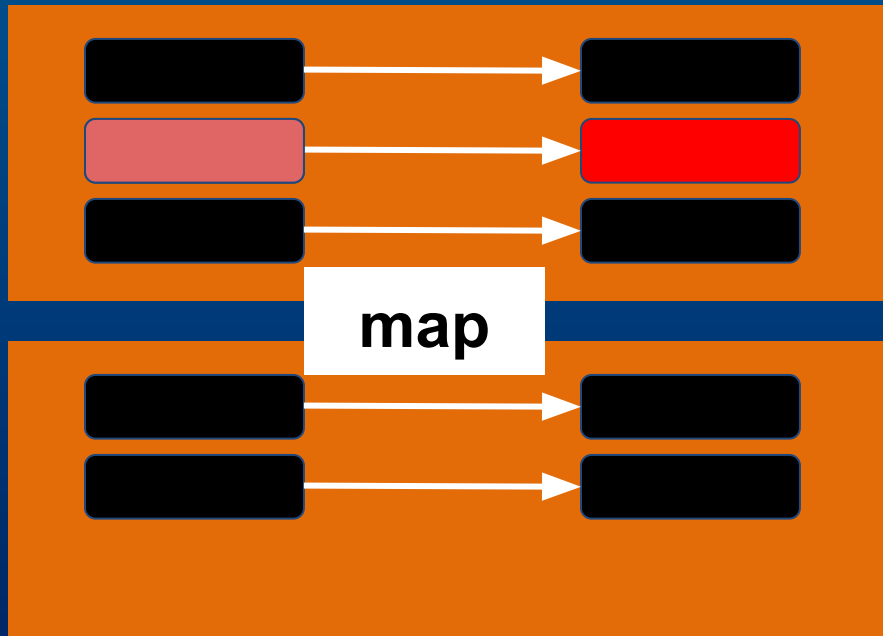
Narrow

vs

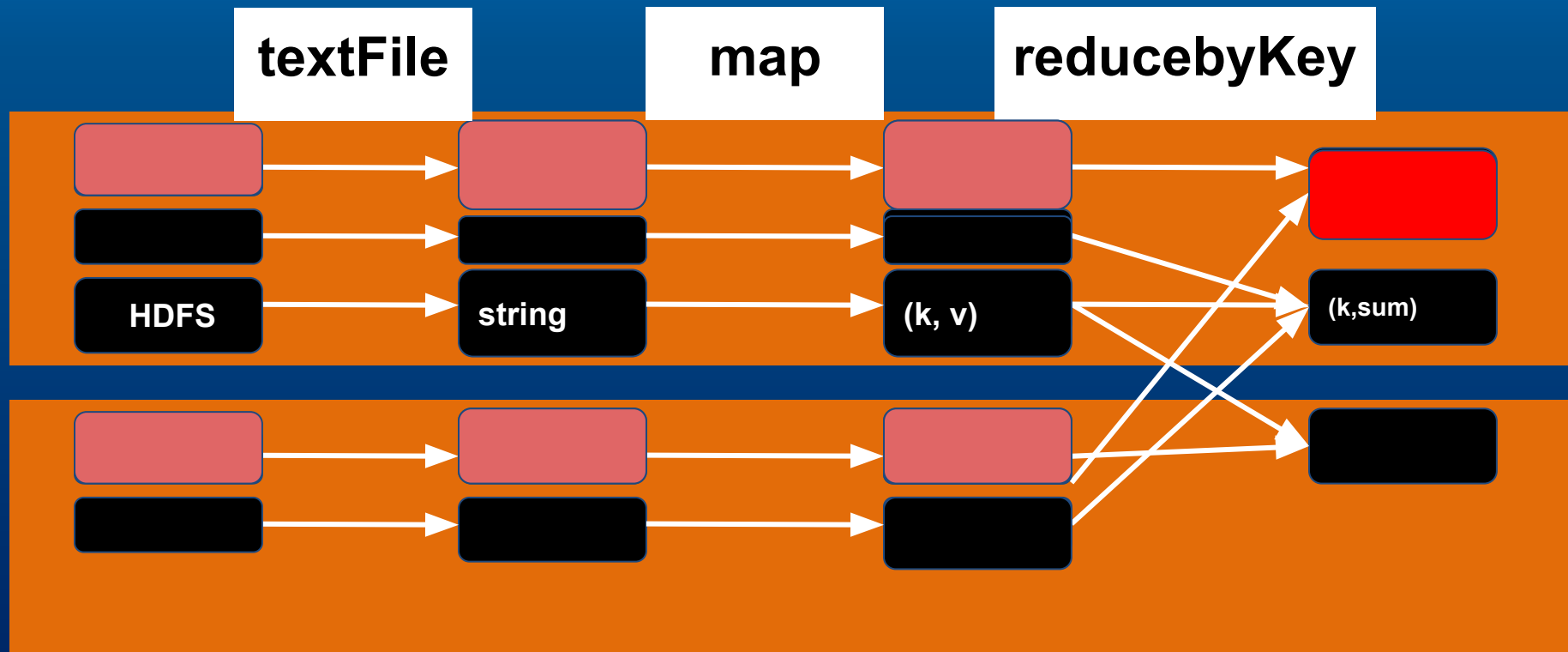
Wide

map

groupByKey

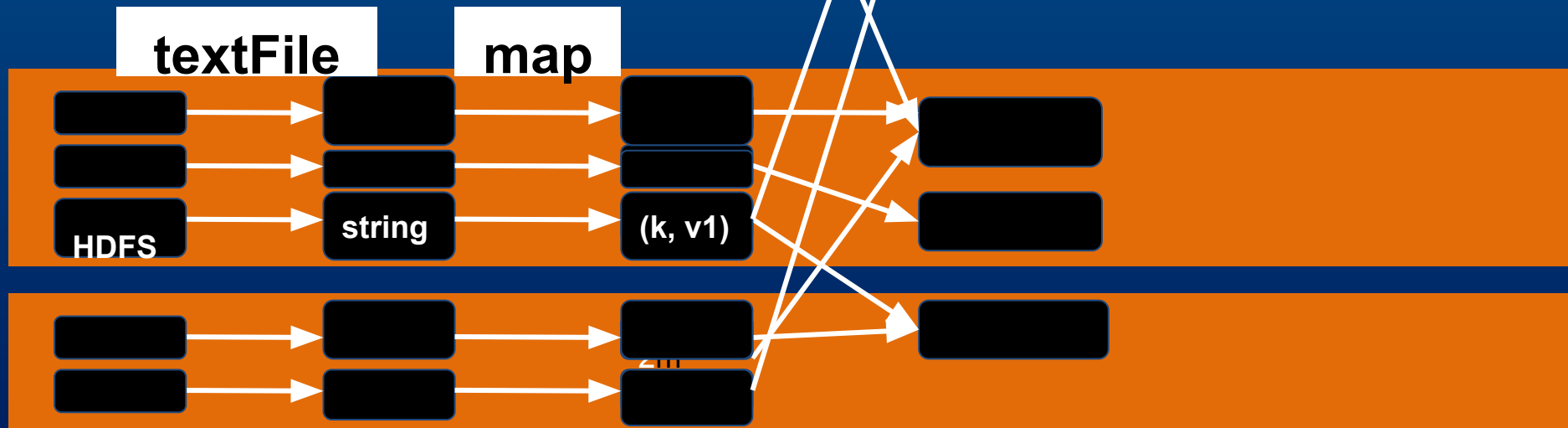
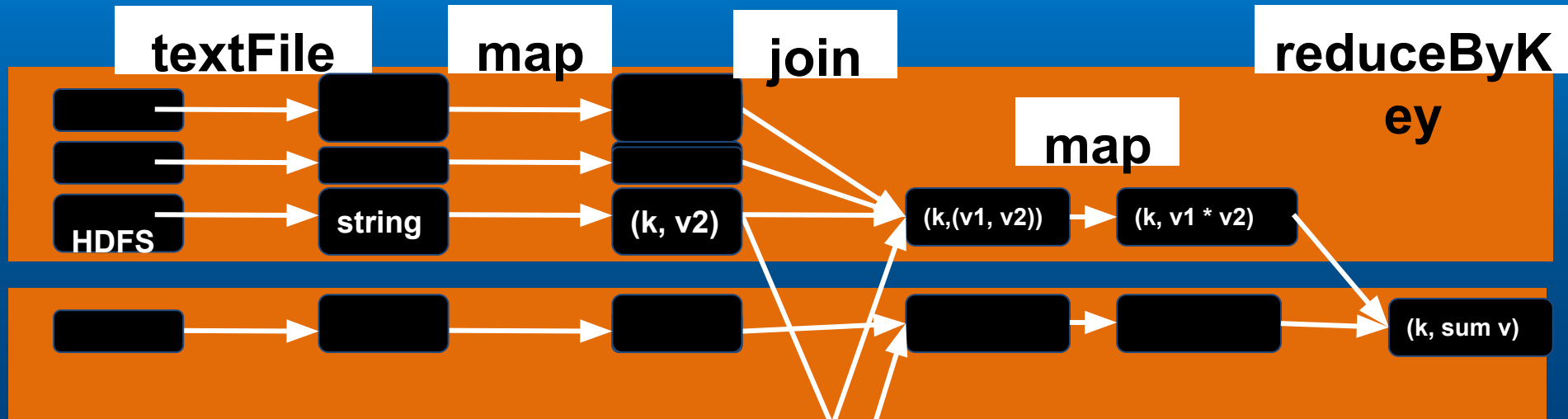


Spark DAG of transformations



House price JOIN

see `house_price_join.ipynb`



Thanks



This work is licensed under
a [Creative Commons
Attribution-ShareAlike 4.0
International License](#).

Questions?

Google Docs

[https://docs.google.com/presentation/d/1wQT7
Bii_ToC2QTk-LCrY6B533-SmsIsIIZ6IjCNdP6U/
edit?usp=sharing](https://docs.google.com/presentation/d/1wQT7Bii_ToC2QTk-LCrY6B533-SmsIsIIZ6IjCNdP6U/edit?usp=sharing)

zonca@sdsc.edu

@andreazonca