Introduction

Introduction (Talking Head, No Slides)

- It's one thing to write some code that works on a toy example, it's another to ensure that it scales efficiently in production! In this Part II of our introduction to Spark, we'll look at ways to improve the performance of our Spark jobs by understanding how the various elements of the framework influence execution.
- We'll then apply these ideas to develop an efficient K-means algorithm at scale.
- While this week is focused specifically on Spark, many of the concepts are universal; diligent application monitoring and debugging, understanding data serialization, garbage collection, and controlling data locality through partitioning—these are all applicable in any distributed context.
- The Spark-specific sections are based heavily on the book *Spark: The Definitive Guide,* which is written by Matei Zacharia, one of the original creators of Spark, and Bill Chambers, a long-time Spark user and product manager at Databricks.

The End

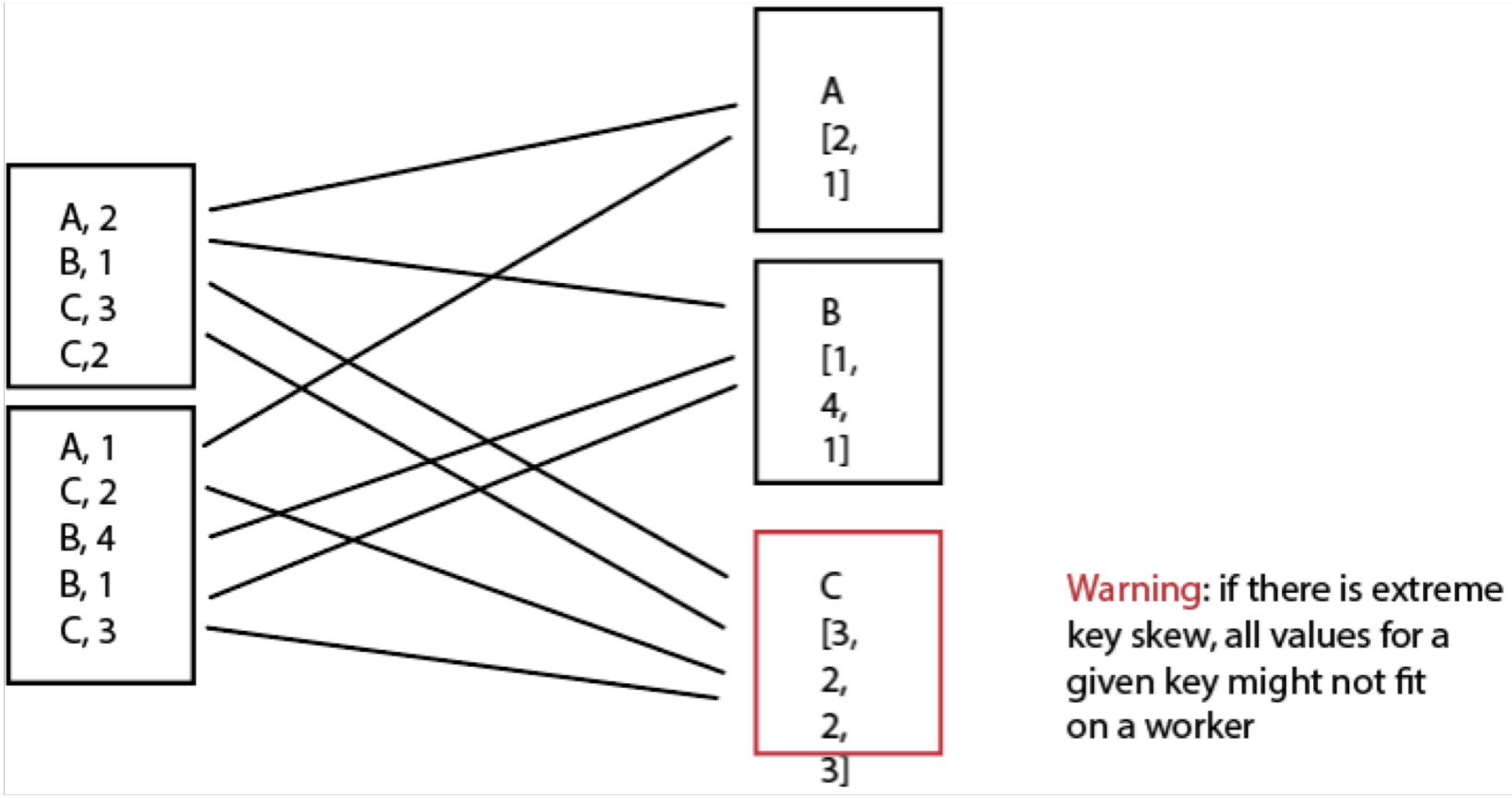
Aggregations

Understanding Aggregation Implementations

- groupByKey
- groupBy
- reduceByKey
- reduceBy

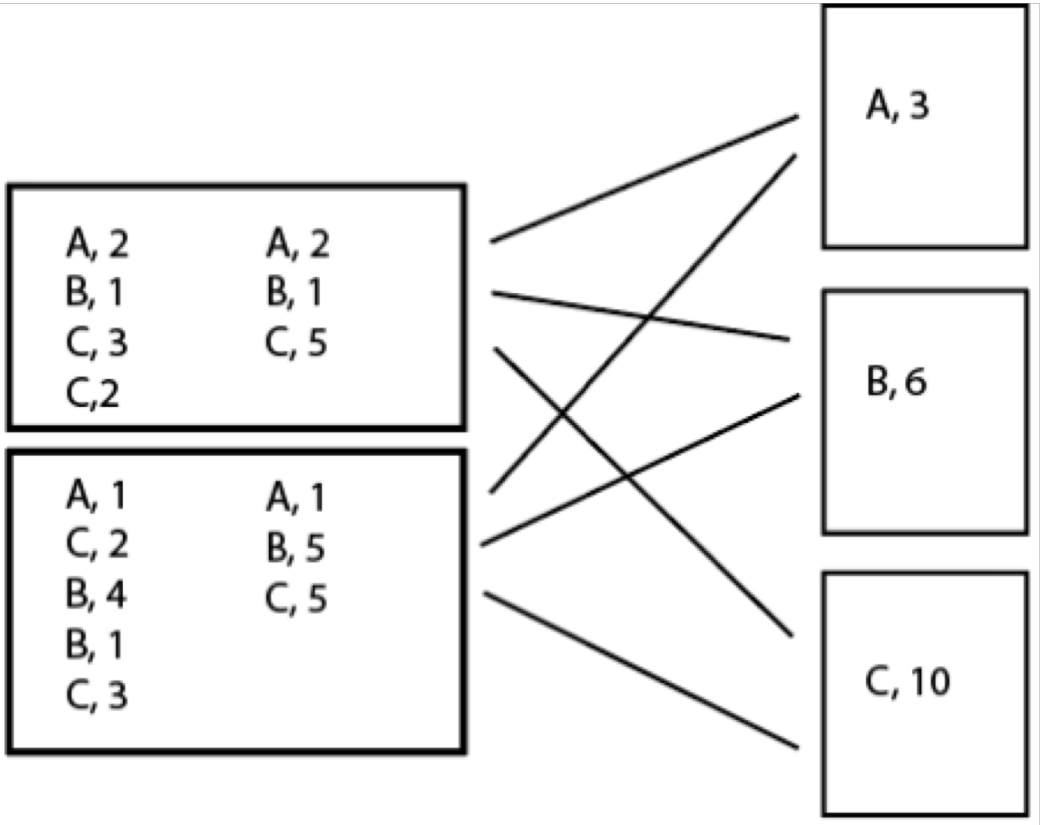
Understanding Aggregation Implementations

groupByKey()



Understanding Aggregation Implementations

reduceByKey(Func)



Performs local aggregation!

Other Aggregation Methods

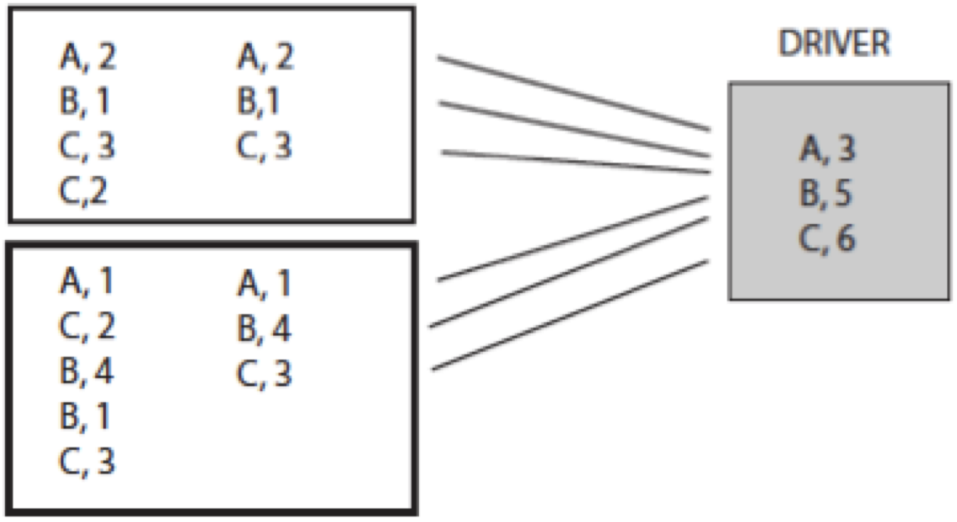
- aggregate
- treeAggregate
- aggregateByKey
- combineByKey
- foldByKey

Understanding Aggregation Implementations

aggregateByKey(zeroValue, seqOp, comb Op)

start value,
seqOp -> within partition function,
combOp -> across partition function

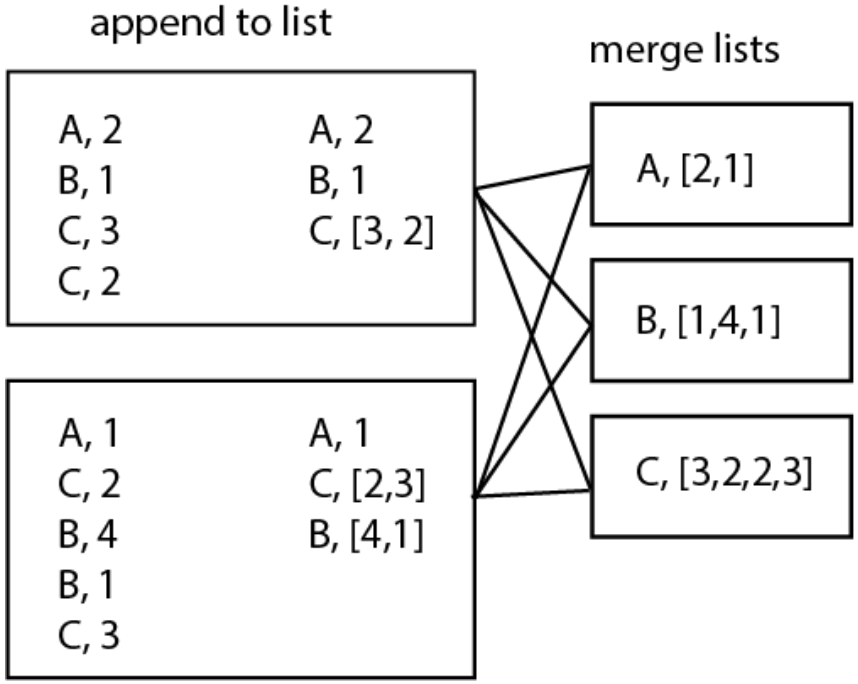
ex: aggregateByKey(0,max,add)



Warning: if the results from the executors are too large, they can take down the driver with an `OutOfMemoryError`

Understanding Aggregation Implementations

combineByKey(createCombiner, mergeValue, mergeCombiners)



CoGroups, Joins, and Zips

- CoGroups
- Joins
- Zips

```
rdd1 = sc.parallelize([('a',1),('b',1)])
rdd2 = sc.parallelize([('a',1),('c',1)])
```

```
rdd1.cogroup(rdd2).collect()
```

```
[('a',
  (<pyspark.resultiterable.ResultIterable at 0x7fac38bc4668>,
   <pyspark.resultiterable.ResultIterable at 0x7fac38bc47b8>)),
 ('b',
  (<pyspark.resultiterable.ResultIterable at 0x7fac38bc49b0>,
   <pyspark.resultiterable.ResultIterable at 0x7fac38bc46d8>)),
 ('c',
  (<pyspark.resultiterable.ResultIterable at 0x7fac38bc4518>,
   <pyspark.resultiterable.ResultIterable at 0x7fac38bc4d68>))]
```

```
rdd1.join(rdd2).collect()
```

```
[('a', (1, 1))]
```

```
rdd1.zip(rdd2).collect()
```

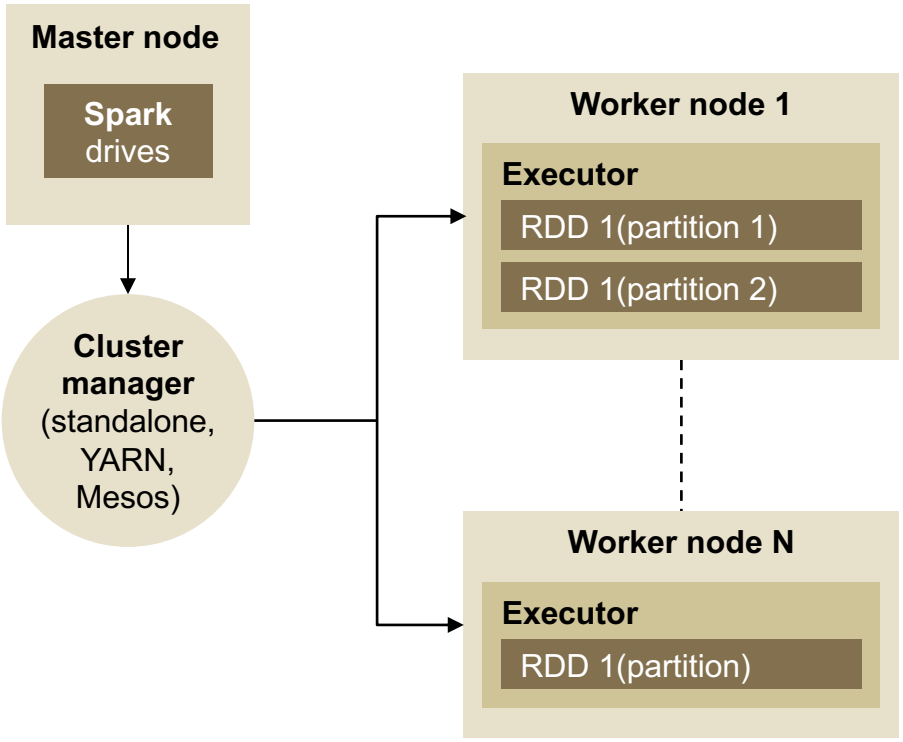
```
[(('a', 1), ('a', 1)), (('b', 1), ('c', 1))]
```

The End

Controlling Partitions

HashPartitioner and RangePartitioner

- Spark structured API leverages **HashPartitioner** for discrete values and **RangePartitioner** for continuous values to derive the partitioning scheme and also how many partitions to create.
- However, unlike the higher-level structured API, RDDs give us more fine-grained control over how the data are partitioned over the network.



Built-In Partitioner Functions

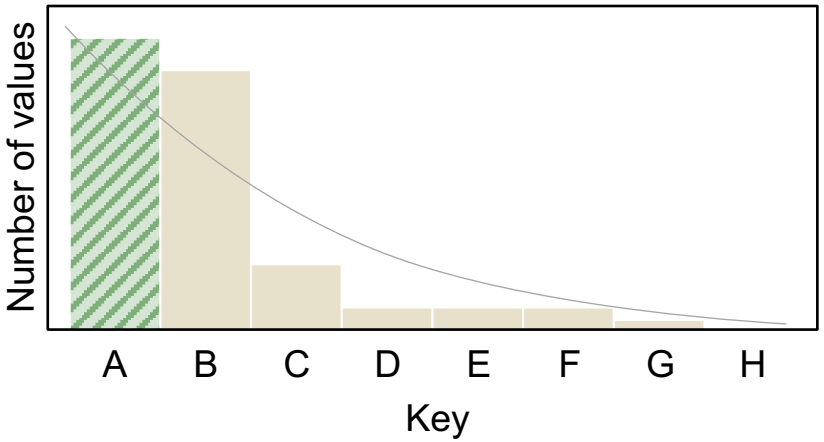
- **coalesce:** collapses partitions on the same worker
- **repartition:** performs a shuffle across nodes
- **repartitionAndSortWithinPartitions:** repartition as well as specify the ordering of each partition *(how might you use this to implement Total Order Sort?)*

```
rdd.coalesce(1).getNumPartitions()
// 1

rdd.repartition(10)
// gives us 10 partitions
```

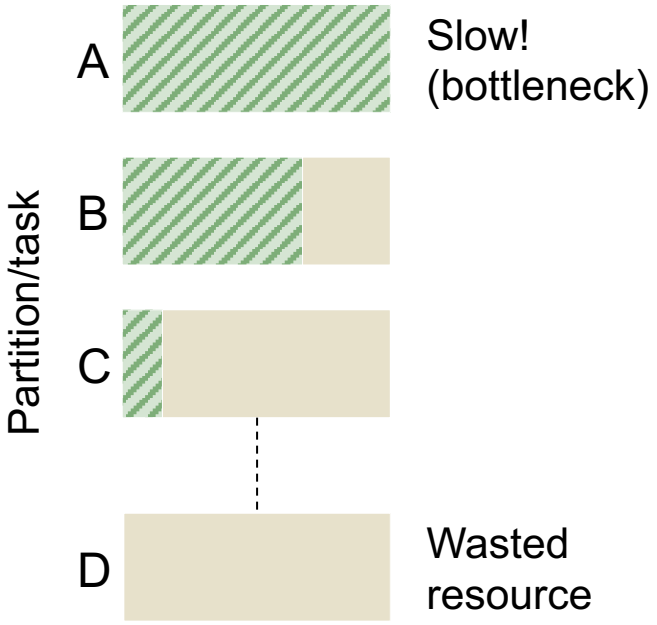
Custom Partitioning

Data skew: We seek to even-out the distribution of the data across the cluster to avoid bottlenecks.

Zipf's distribution

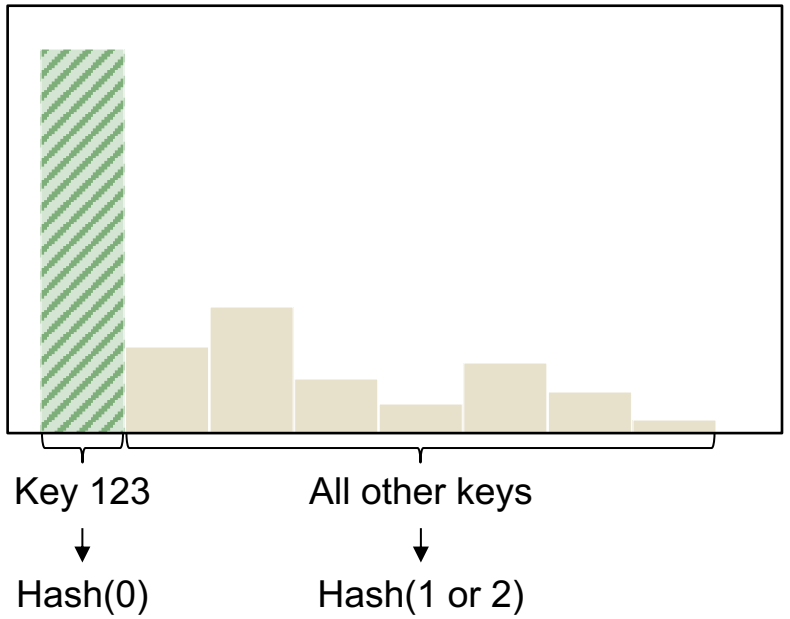


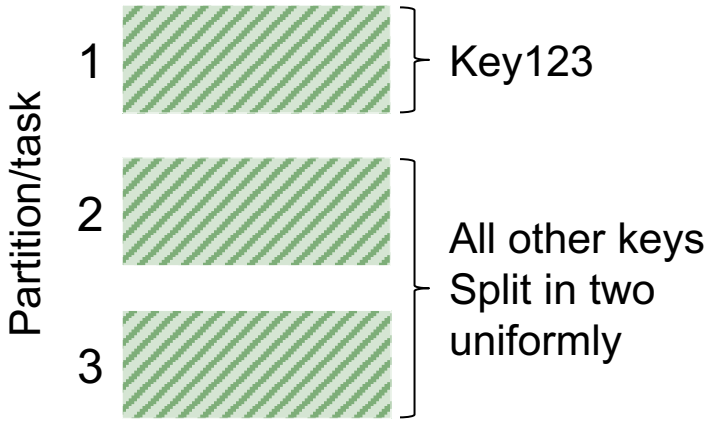
Naive partitions



Example

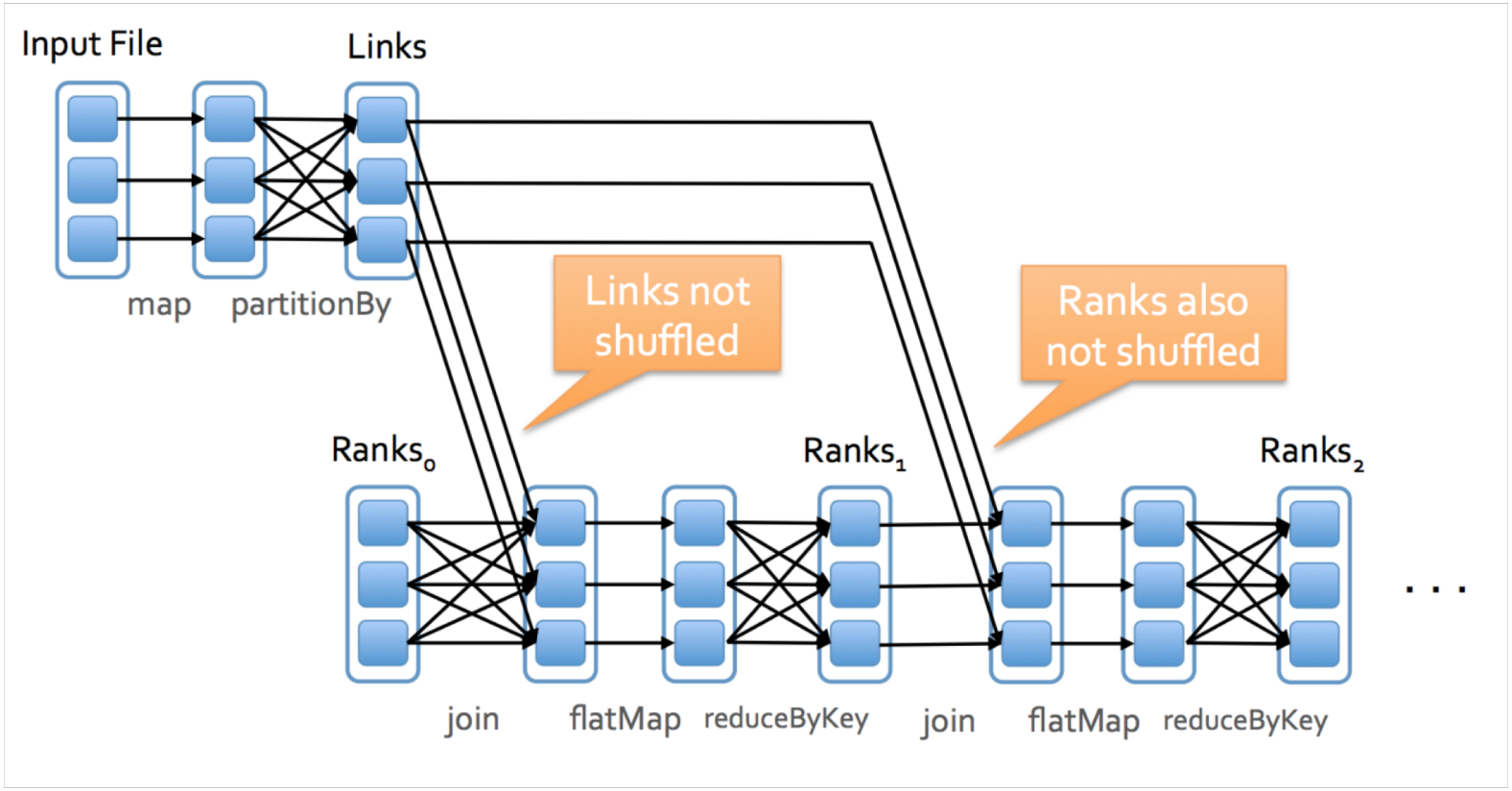
```
def partitionFunc(key):
        import random
        if key == 123:
                return 0
        else:
                return
random.randint(1,2)

rdd.partitionBy(3, partitionFunc)\
        .map(lambda x: x[0])\
        .glom()\
        .map(lambda x:
len(set(x)))\
        .collect()

// [1, 4294, 4312]
```





A Case for RDDs

PageRank: We seek to control the layout of the data on the cluster to avoid shuffles.

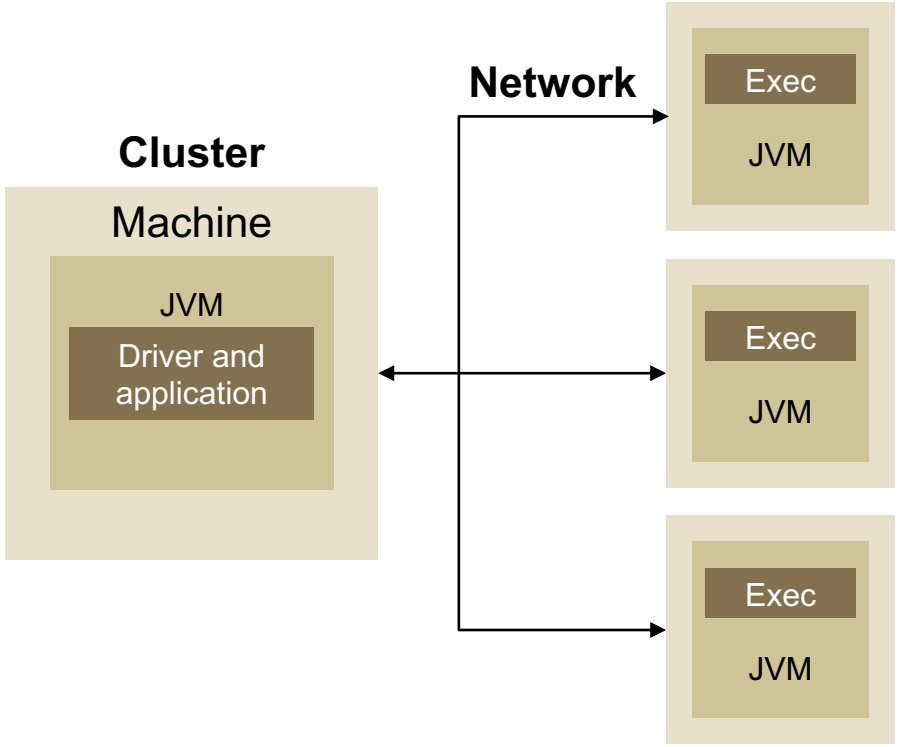


The End

Monitoring and Debugging

The Monitoring Landscape

- **Spark applications** and **jobs**: Spark UI and Spark logs
- **JVM**: stack traces, low-level tools to help profile Spark jobs
- **OS/Machine**: CPU, network, I/O
- **Cluster** (YARN, Mesos, standalone): *Ganglia, Prometheus*



Components of a spark application that you can Monitor
Figure 18-1: *Spark: The Definitive Guide*

What To Monitor

- Processes: CPU usage, memory usage
- Query execution: jobs, tasks
  - Spark Logs
  - Spark UI

Spark Logs

- Using Python's `logging` module
- Messages are written to **stderr** when running a local mode application
- Messages are written to **log files** by your cluster manager when running Spark on a cluster

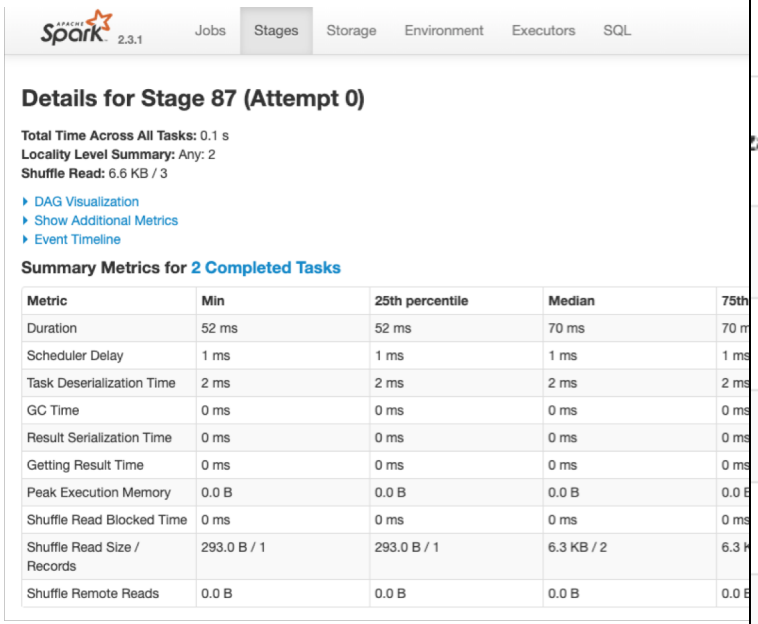
```
spark.sparkContext.setLogLevel("INFO")
```

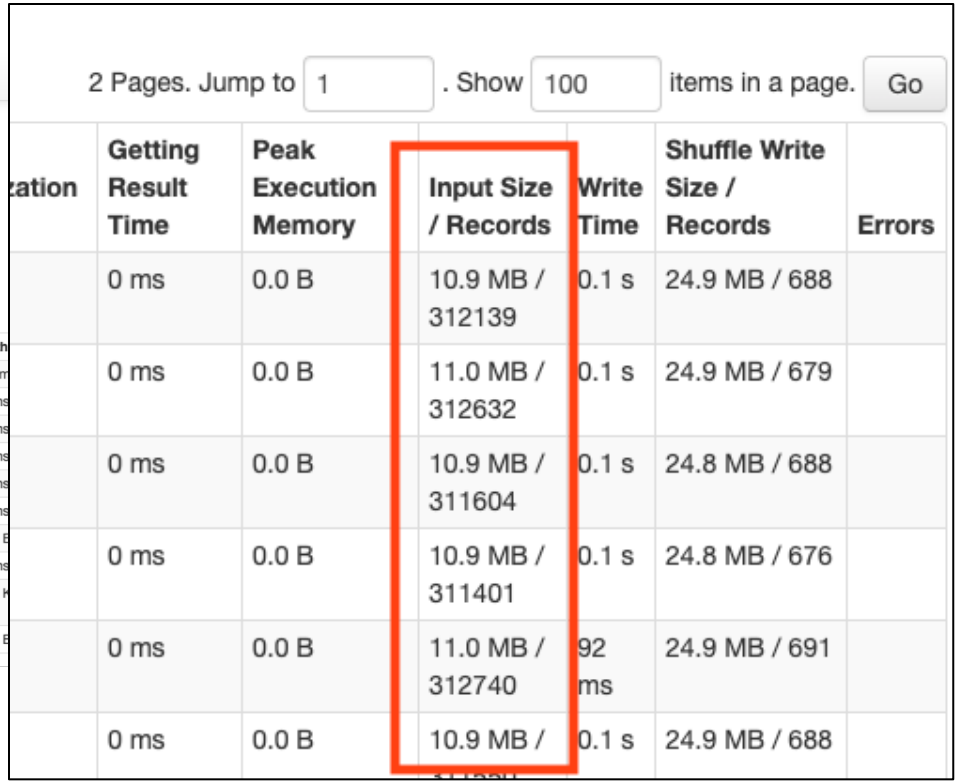
```
import logging

logging.warning('Watch out!')  # will print a message to the console

logging.info('I told you so')  # will not print anything
```

Logging HOWTO

The Spark UI

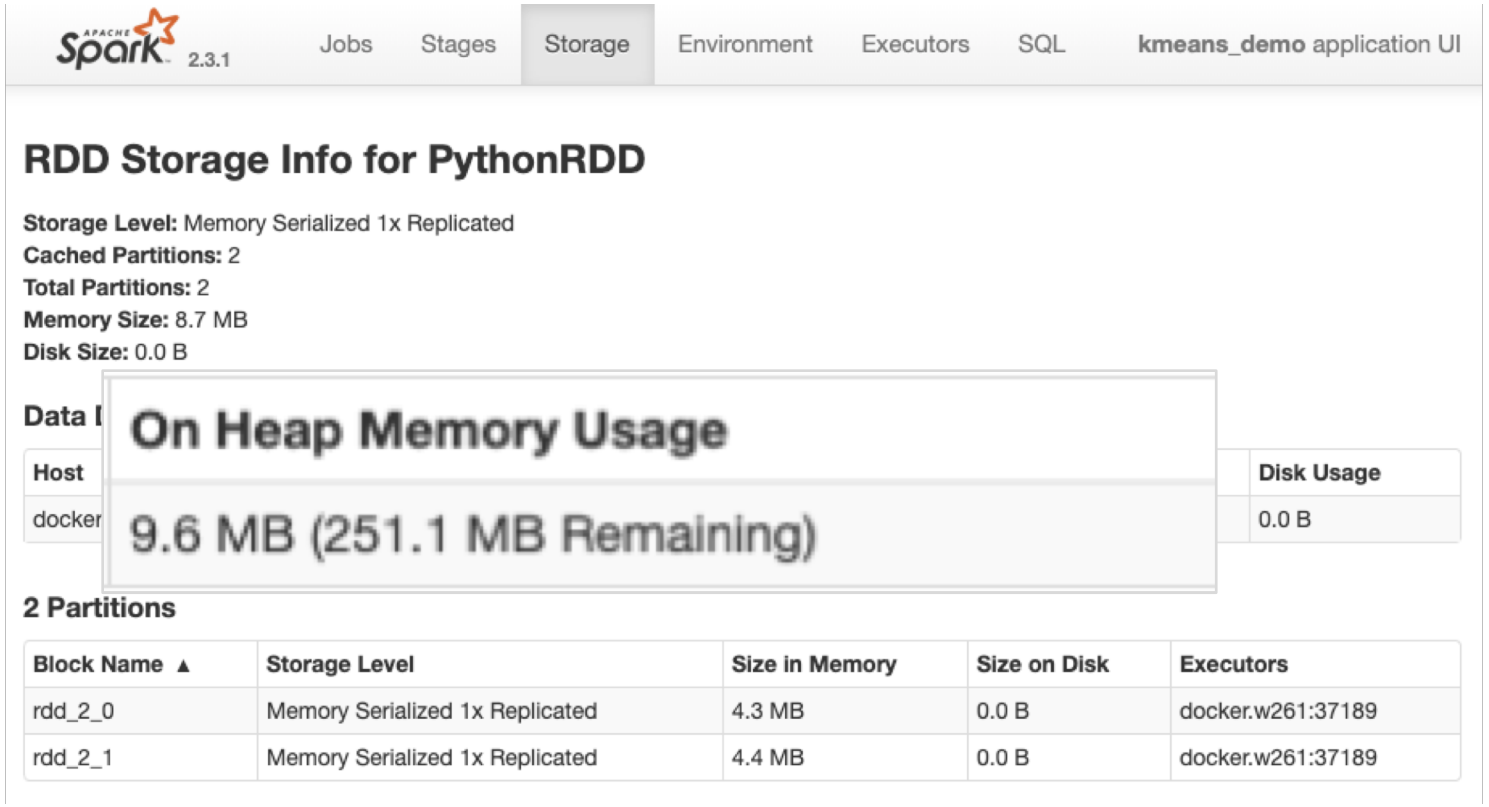
- **Summary metrics** for completed tasks—look out for uneven distributions!
- Storage tab: gives insight to garbage collection
- Environment tab: your Spark configuration





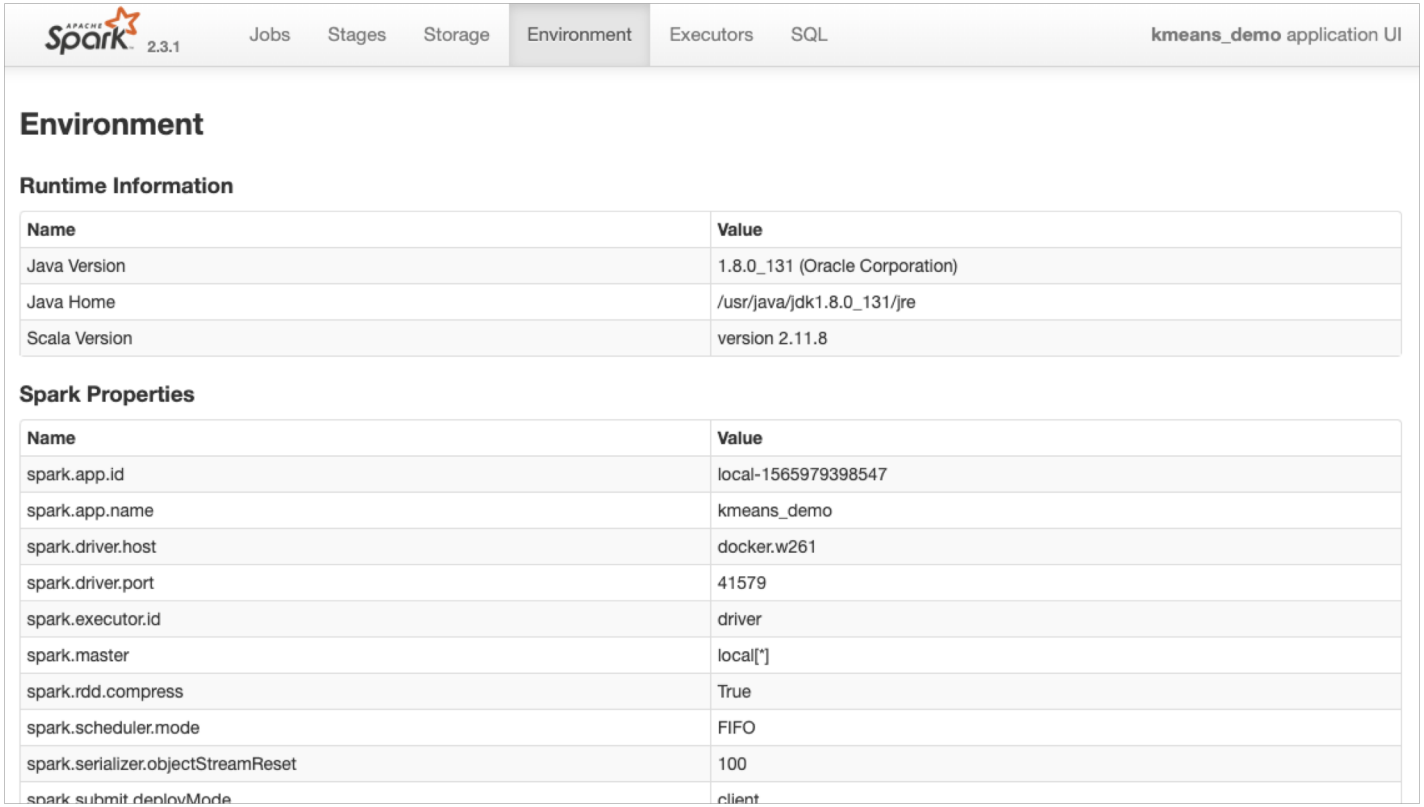
The Spark UI

- Summary metrics for completed tasks—look out for uneven distributions!
- **Storage tab:** gives insight to garbage collection
- Environment tab: your Spark configuration



The Spark UI

- Summary metrics for completed tasks—look out for uneven distributions!
- Storage tab: gives insight to garbage collection
- **Environment tab:** your Spark configuration



Spark REST API

So you can build your own reporting tools!

`http://localhost:4040/api/v1`

Spark UI History Server

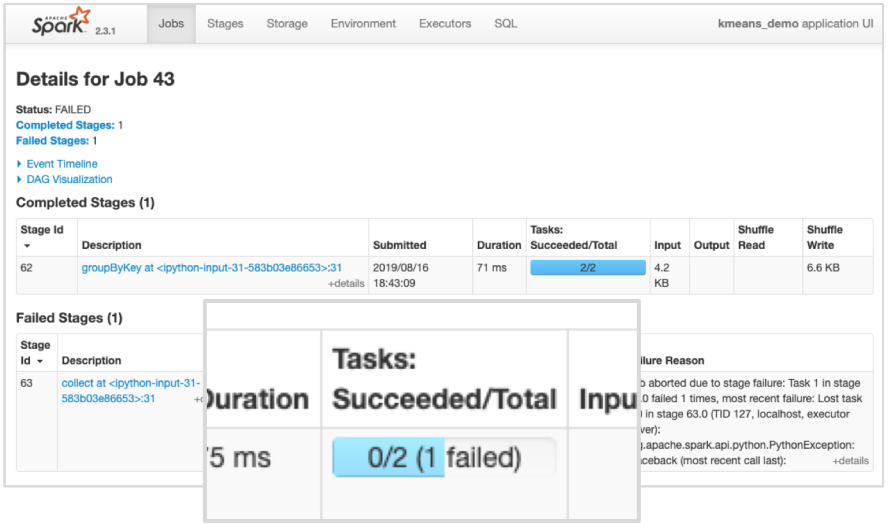
- How can we get to the Spark UI after an application ends or crashes?
- Use the history server by enabling event logs.

```
spark.eventLog.enabled

spark.eventLog.dir
```

The End

Spark First Aid

Debugging and Spark First Aid



Spark jobs not starting

Symptoms

- Spark jobs don’t start
- Spark UI doesn’t show any nodes except the driver
- Spark UI reporting wrong information

Treatments

- Open all ports between the worker nodes
- Ensure Spark resource configurations are correct.

Errors before execution

Symptoms

- Commands don’t run at all and output large error messages
- In the Spark UI, no jobs, stages, or tasks seem to run

Treatments

- look for programmer errors—typos, incorrect file paths, class paths, etc…

Errors during execution

Symptoms

- One job run but the next one fails
- A step in a multi-step query fails
- A scheduled job that ran yesterday fails today
- Difficult to parse error message

Treatments

- check the stack trace
- check that your data are formatted as expected
- look for failed tasks in the Spark UI
- check the logs for those tasks
- add more logging inside your code

Slow Tasks or Stragglers

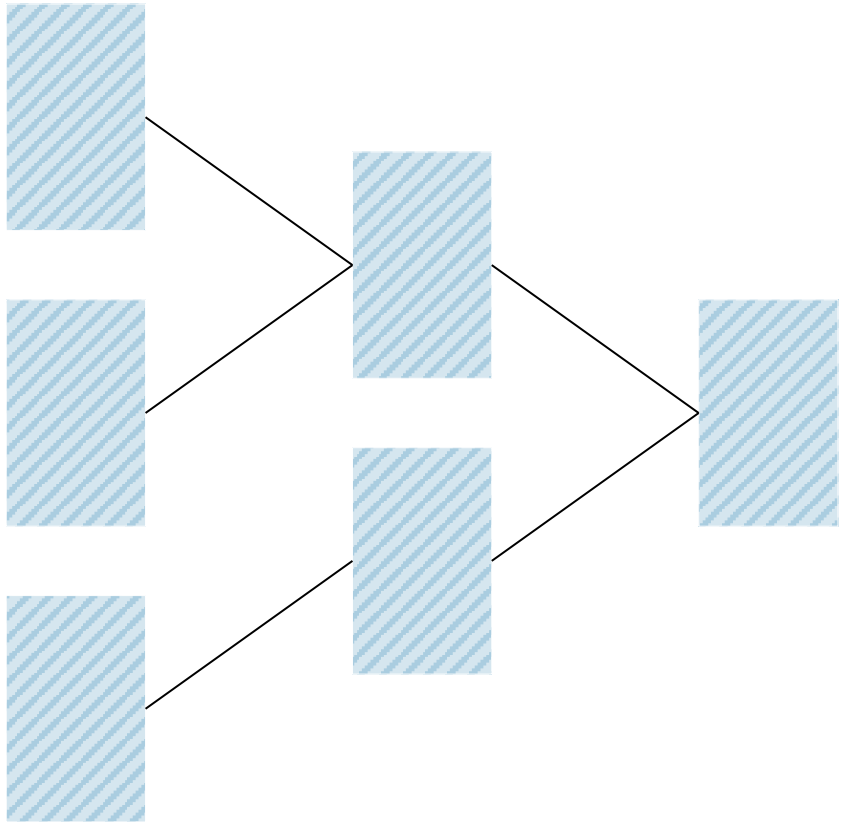
- When scaling up, number of machines doesn't help
- When certain executors are reading and writing much more data than others
- **"Stragglers"** are often due to data being unevenly partitioned

- Did you use a groupByKey?
- Do you have wasteful UDFs or UDAFs pulling a lot of data into memory?
- Do you have faulty nodes? Try turning on speculation*

**More about speculation and trade-offs later*

Slow Aggregations

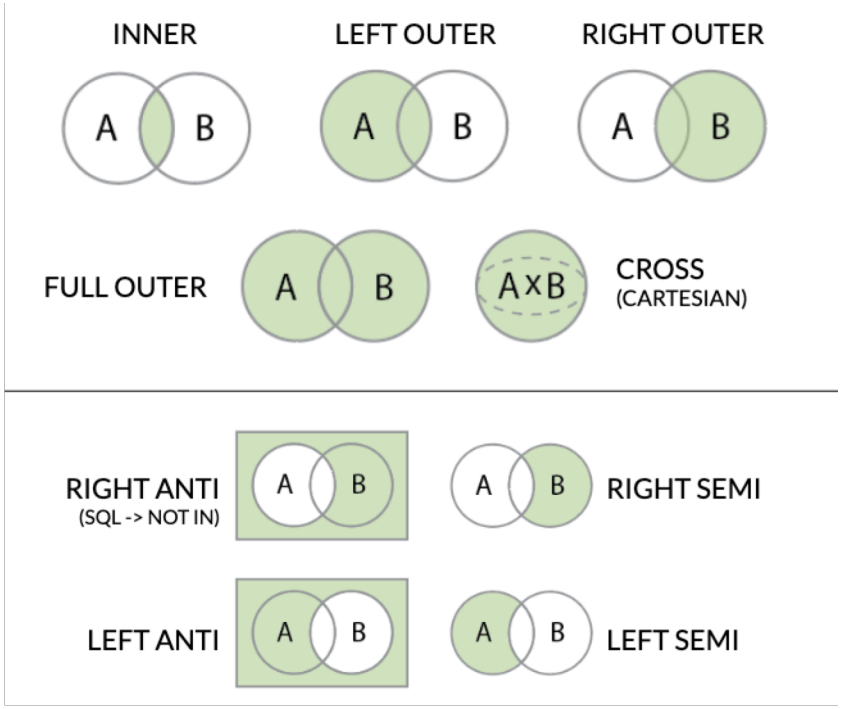
- Sometimes slow can't be helped; the data have some skewed keys and the operation you want to run on them needs to be slow.
- Things to try
  - Increase number of partitions
  - Increase executor memory
  - Repartition after aggregating
  - Filter first!
  - Use null instead of your own custom representations for null values



Slow Joins

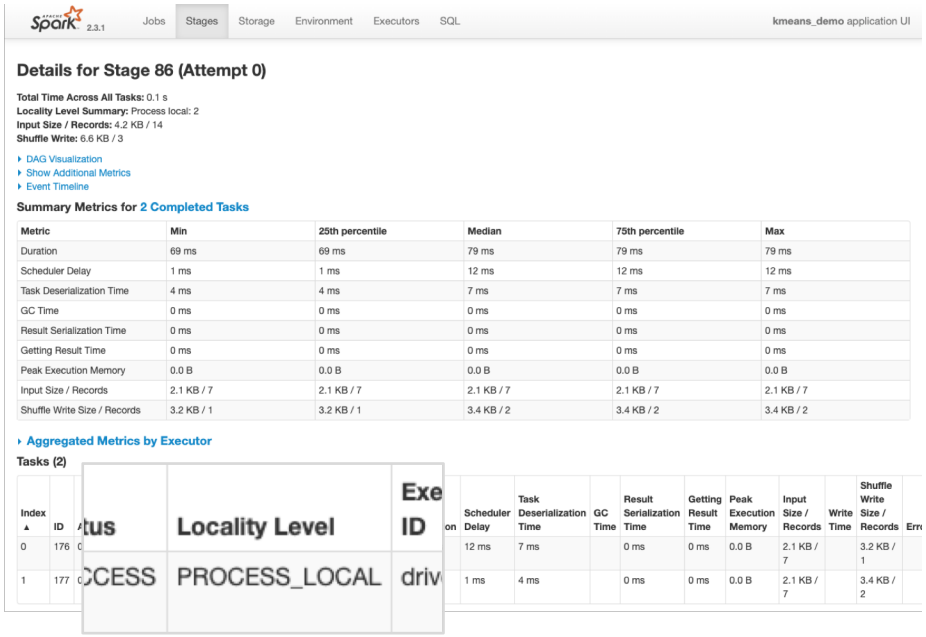
- A join stage is taking a long time yet stages before and after the join seem to be operating normally
- Things to try
  - Experiment with join types
  - Join ordering
  - Partition before joining
  - Filter and select before join
  - Force broadcast join if necessary

More on joins in week 8!



Slow Reads and Writes

- Slow I/O can be difficult to diagnose, especially with networked file systems
- Things to try
  - Turn on speculation—but beware of data duplication!
  - Ensure that network bandwidth is sufficient
  - Enable Spark locality-aware scheduling



Driver OutOfMemoryError

Symptoms

- Spark application unresponsive or crashed
- `OutOfMemoryErrors` or garbage collection messages in the driver logs
- Commands take a long time to run or don't run at all
- Interactivity is slow or nonexistent
- Memory usage is high for the driver JVM

Treatments

- Did you try to `collect()` a large amount of data?
- Is your broadcast join table too big?
- Heap is full?
- Are you sharing a `SparkContext` with other users?

Other Errors

- Executor OutOfMemoryError
- Unexpected nulls in results
- No space left on disk errors
- Serialization errors

Conclusion

- Step-by-step approach
- Add logging
- Isolate the problem to smallest piece of code possible
- Use Spark's UI
- **Happy debugging!**

The End

Performance Tuning

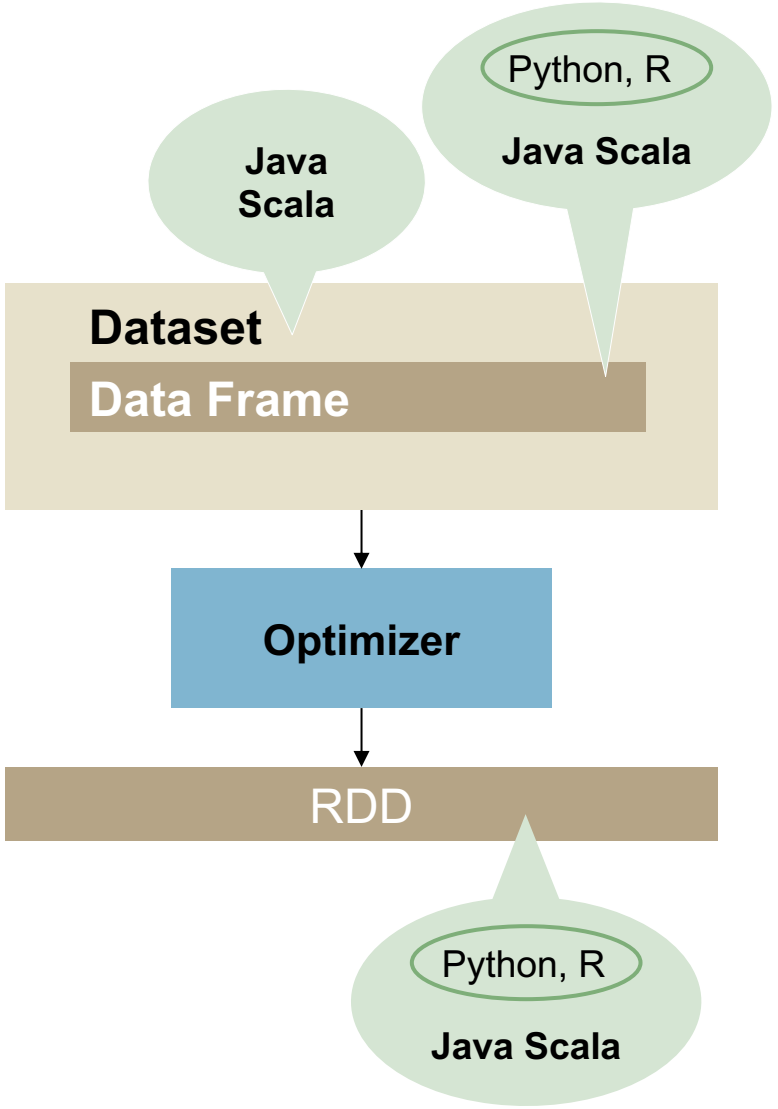
What Can We Optimize

- Code-level design choices (RDD vs. DataFrame)
- Data at rest
- Joins
- Aggregations
- Data in flight
- Individual application properties
- Inside the JVM of an executor
- Worker nodes
- Cluster and deployment properties

- **Indirect performance enhancements:** configuration settings
- **Direct performance enhancements:** design choices at the individual job level

Indirect Performance Enhancements

- **Design choices**
- Object serialization in RDDs
- Cluster configuration
- Scheduling
- Data at rest
- Shuffle configurations
- Memory pressure and garbage collection



Indirect Performance Enhancements

- Design choices
- **Object serialization in RDDs**
- Cluster configuration
- Scheduling
- Data at rest
- Shuffle configurations
- Memory pressure and garbage collection

Indirect Performance Enhancements

- Design choices
- Object serialization in RDDs
- **Cluster configuration**
- Scheduling
- Data at rest
- Shuffle configurations
- Memory pressure and garbage collection

Indirect Performance Enhancements

- Design choices
- Object serialization in RDDs
- Cluster configuration
- **Scheduling**
- Data at rest
- Shuffle configurations
- Memory pressure and garbage collection

Indirect Performance Enhancements

- Design choices
- Object serialization in RDDs
- Cluster configuration
- Scheduling
- **Data at rest**
- Shuffle configurations
- Memory pressure and garbage collection
- Table partitioning
- Bucketing
- Number of files
- Data locality
- Statistics collection

Indirect Performance Enhancements

- Design choices
- Object serialization in RDDs
- Cluster configuration
- Scheduling
- Data at rest
- **Shuffle configurations**
- Memory pressure and garbage collection

Indirect Performance Enhancements

- Design choices
- Object serialization in RDDs
- Cluster configuration
- Scheduling
- Data at rest
- Shuffle configurations
- **Memory pressure and garbage collection**

- Collecting statistics on garbage collection
  - `-verbose:gc`
  - `-XX:+PrintGCTimeStamps`
- Garbage collection tuning

Direct Performance Enhancements

- Parallelism
- Improved filtering
- Partitioning and coalescing
- User-defined functions (UDFs)
- Temporary data storage (caching)
- Joins
- Aggregations
- Broadcast variables

Conclusion

- Read as little data as possible through partitioning and efficient binary formats.
- Make sure there is sufficient parallelism and no data skew on the cluster using partitioning.
- Use the structured APIs as much as possible to avail of optimized code.

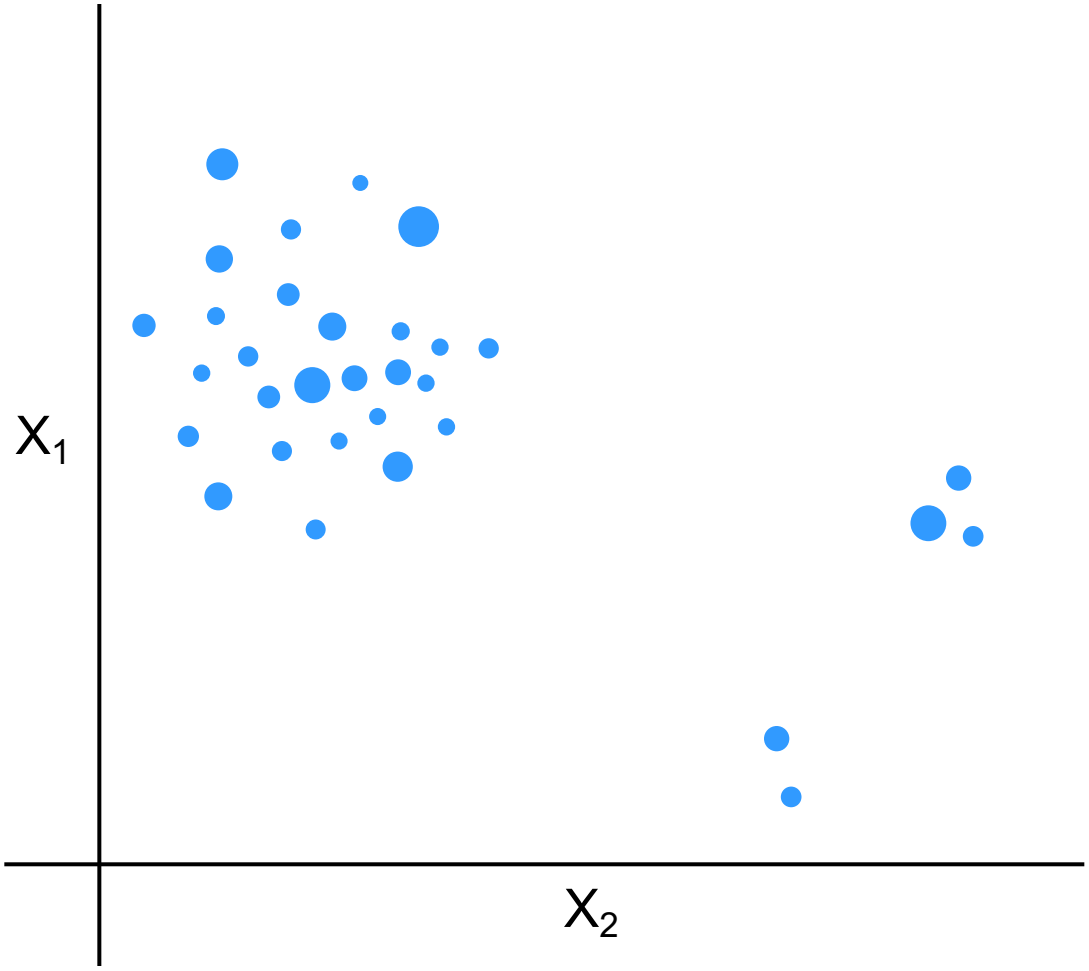
The End

K-Means at Scale

A Case for Spark

- K-means is an iterative algorithm consisting of many passes over the data; as we saw last week, this is an area where Hadoop MapReduce really falls down and Spark shines. Hadoop requires that we write our entire data set out to disk at every iteration, making this type of algorithm incredibly inefficient.
- Let's take advantage of some of the performance tuning techniques from this week to see if we can implement an efficient K-means model in Spark.

Dataset



Version 1

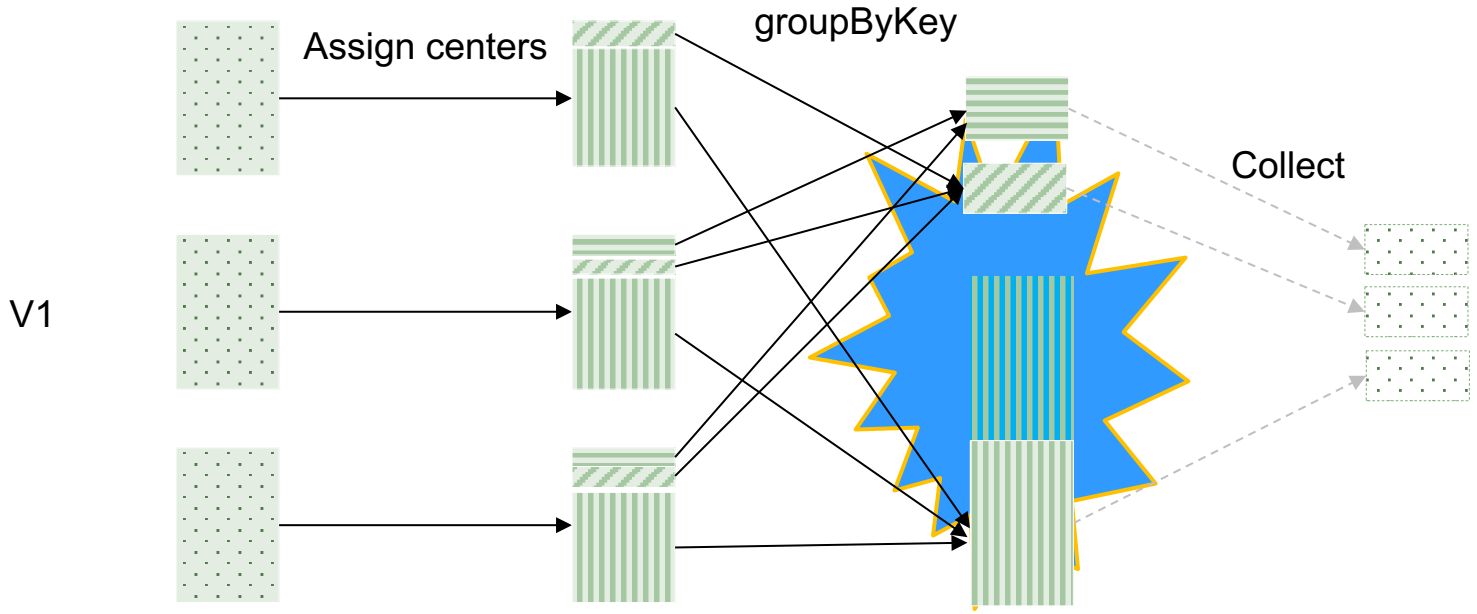
1. Set K = 3
2. Broadcast centers
3. RDD.map(*assignCentersByDistance*)
   .groupByKey()
   .map(*calculateAvg*)
   .collect()

Repeat steps 2 and 3 until convergence

Version 1 Illustrated

All values for a given key are sent to the same reducer.

A reducer can run out of memory if there are too many values associated with a given key.

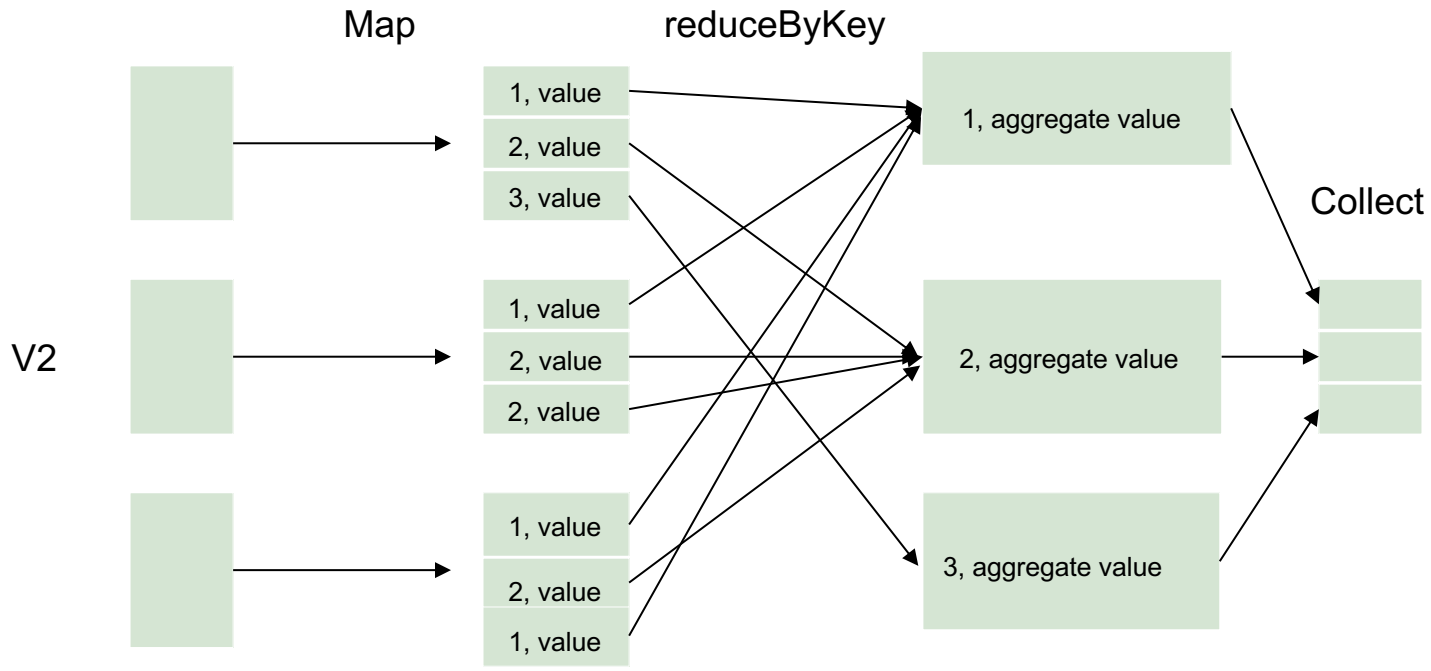


Version 2

1. Set K = 3
2. Broadcast centers
3. RDD.map(*assignCentersByDistance*)
   .reduceByKey(*calculateSumsAndCounts*)
   .map(*calculateAvg*)
   .collect()

Repeat steps 2 and 3 until convergence

Version 2 Illustrated



Only send aggregated information to reducers; maximum values in each reducer equals number of mappers times K (number of clusters).

Shuffle Write

The groupByKey implementation/Iteration:

- Total Time Across All Tasks: 3 s
- Locality Level Summary: Process local: 2
- Input Size / Records: 2.1 MB / 100000
- Shuffle Write: 3.6 MB / 4

The reduceByKey implementation/Iteration

- Total Time Across All Tasks: 3 s
- Locality Level Summary: Process local: 2
- Input Size / Records: 2.1 MB / 100000
- Shuffle Write: 965.0 B / 4

Can We Do Better?

- Parquet ?
- Partitioning ?
- Caching ?

The End

Summary

<Talking Head, No Slides>

- Today, we looked at some advanced techniques for debugging and optimizing Spark applications; in particular, we implemented several versions of the basic K-means algorithm using RDDs and different aggregation methods.
- We saw that K-means is NP hard if we were to try to find the optimal solution, and, as such, the iterative Lloyd algorithm we implemented is not guaranteed to converge on the global minimum; depending on the initialization of the centroids, results will vary.
- A lot of work has been done on improving the initialization including K-means++ and its parallel variant K-means||, which has been shown to improve both accuracy and time to convergence; more on that and other extensions in our live session!

The End