

# ASync Lecture Slides

---

# Introduction to Spark: Part II

---

## Table of contents

- 5.1 Introduction
- 5.2 Aggregations
- 5.3 Controlling partitions
- 5.4 Monitoring and debugging
- 5.5 Performance tuning
- 5.6 Clustering overview
- 5.7 K-means algorithm
- 5.9 Summary

ASYNC Lecture Slides

---

# The End

# 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.

Introduction

---

# The End

# Aggregations

---

## Understanding Aggregation Implementations

---

- `groupByKey`
- `groupBy`
- `reduceByKey`
- `reduceBy`

Pg 229 Definitive Guide.

Let's talk about aggregations.

There are several ways to bring together your key-value PairRDDs; however, the implementation is actually quite important for job stability. Let's compare two fundamental choices: `groupBy` and `reduce`.

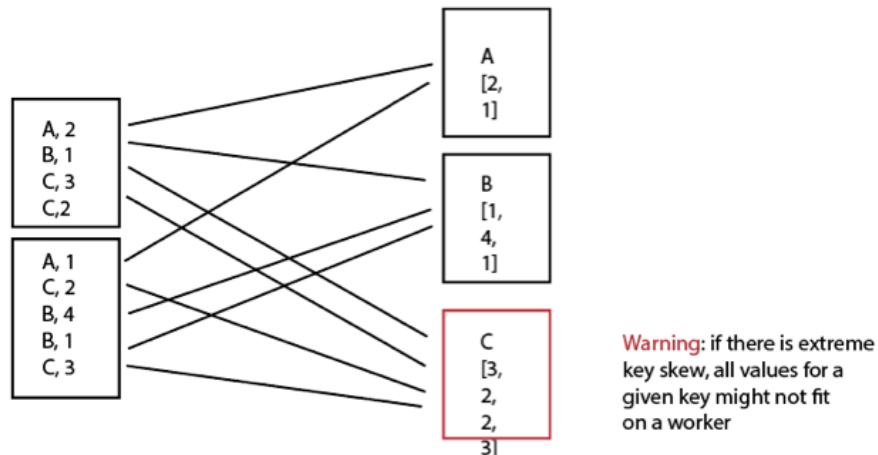
### GroupByKey

Looking at the API, it might seem like using `groupByKey` and then mapping over each grouping is a good way to sum up all the counts for each key.



## Understanding Aggregation Implementations

### groupByKey()



Pg 229 Definitive Guide.

#### GroupByKey

Looking at the API, it might seem like using groupByKey and then mapping over each grouping is a good way to sum up all the counts for each key.

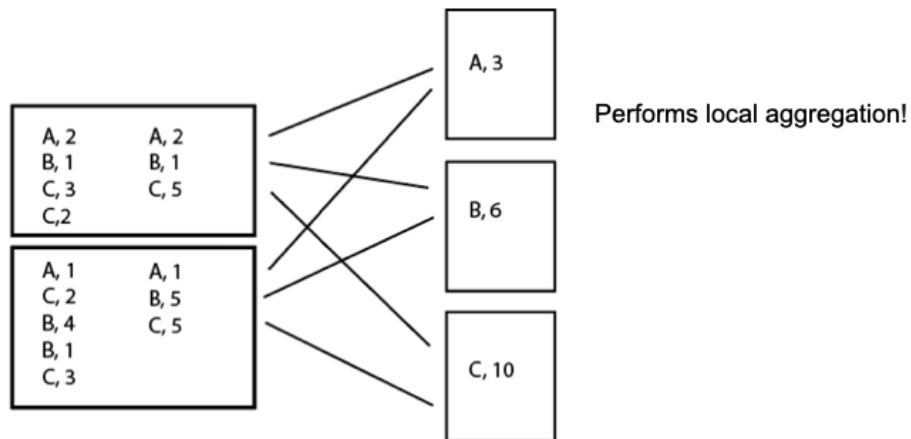
However, for the majority of cases, this is the wrong approach. The fundamental issue here is that each executor must hold all values for a given key in memory before applying the function to them.

This is problematic if you have massive key skew, where some partitions might be completely overloaded with a ton of values and you could get OutOfMemory errors.

OK when you have similar value sizes across keys, and you know they will fit in memory.

## Understanding Aggregation Implementations

### reduceByKey(Func)



A much more stable approach to additive problems is `reduceByKey`. This is because the reduce happens within each partition and doesn't need to put everything in memory. Additionally, there is no incurred shuffle during this operation; everything happens at each worker individually before performing the final reduce.

## Other Aggregation Methods

---

- `aggregate`
- `treeAggregate`
- `aggregateByKey`
- `combineByKey`
- `foldByKey`

Sometimes you'll need more fine grained control over the aggregation process, in which case you can use some of these advanced aggregation functions.

`Aggregate` requires a null and start value and then requires you to specify two different functions. The first aggregates within partitions, and the second aggregates across partitions. The start value will be used at both aggregation levels. `Aggregate` does have some performance implications because it performs the final aggregation on the driver. As you can probably imagine at this point, if the results from the executors are too large, they can take down the driver with an `OutOfMemory` error.

There is another method, `treeAggregate` that does the same thing as `aggregate` (at the user level) but does so in a different way. It basically "pushes down" some of the sub-aggregations (creating a tree from executor to executor) before performing the final aggregation on the driver.

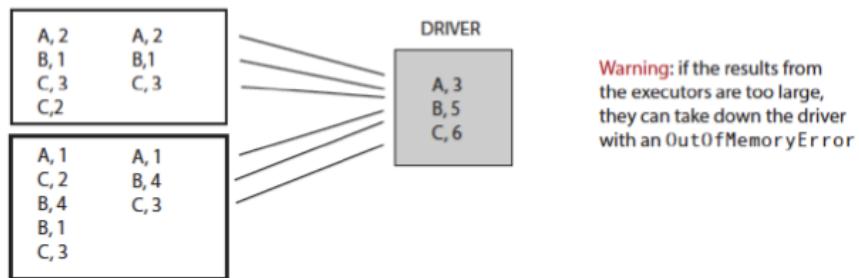
`aggregateByKey` does the same thing as `aggregate`, but instead of doing it partition by partition, it does it by key.

# Understanding Aggregation Implementations

## [aggregateByKey\(zeroValue, seqOp, comb Op\)](#)

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

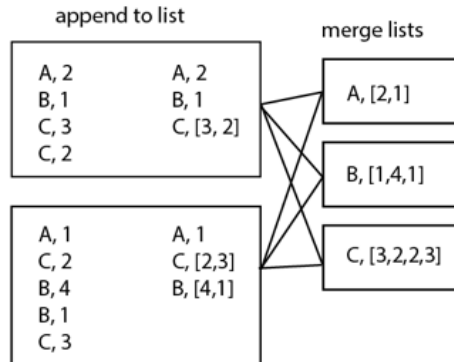
ex: `aggregateByKey(0,max,add)`



An example of when you might want to use aggregate is if you say wanted to calculate the average of the max values of each partition.  
(talk through example)

## Understanding Aggregation Implementations

[combineByKey\(createCombiner, mergeValue, mergeCombiners\)](#)



We can use a `combineByKey` function to achieve similar functionality, but this time the results are NOT brought back to the driver, and, as such, we can specify the number of output partitions.

The first function input to the combiner specifies how to merge values, and the second function specifies how to merge combiners. For example, we might want to add values to a list, and subsequently merge the lists.

# 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)) ]
```

In addition to these summative type aggregations, we also have some join type aggregations.

CoGroups give you the ability to group together up to three key-value RDDs together in Scala, and two in Python. This joins the given values by key. This is effectively just a group-based join on an RDD. Again, you can specify the number of output partitions or a custom partition function to control exactly how this data is distributed across the cluster.

There are all the typical joins such as inner join, full outer join, left outer join, right outer join, and cartesian (watch out with this one! More on that later)

And there is also a zip function which isn't technically a join, but does bring together two RDDs.

Next we'll talk about partitioning.

Aggregations

---

**The End**

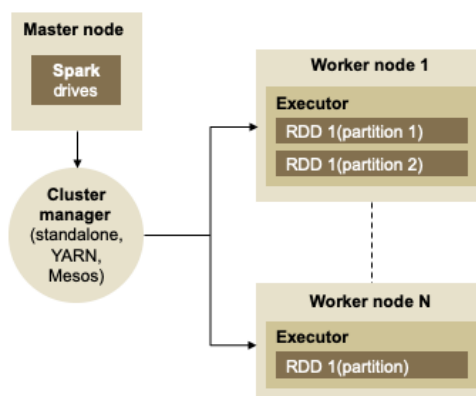
# Controlling Partitions

---



# 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.



<https://labs.criteo.com/2018/06/spark-custom-partitioner/>

<http://sparkdatasourceapi.blogspot.com/2016/10/partitioning-in-spark-writing-custom.html>

<https://www.slideshare.net/SparkSummit/handling-data-skew-adaptively-in-spark-using-dynamic-repartitioning>

Spark has two built-in Partitioners that you can leverage in the RDD API, a HashPartitioner for discrete values, and a RangePartitioner for continuous values. The structured APIs leverage these already. Although useful, they are fairly rudimentary.

## 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
```

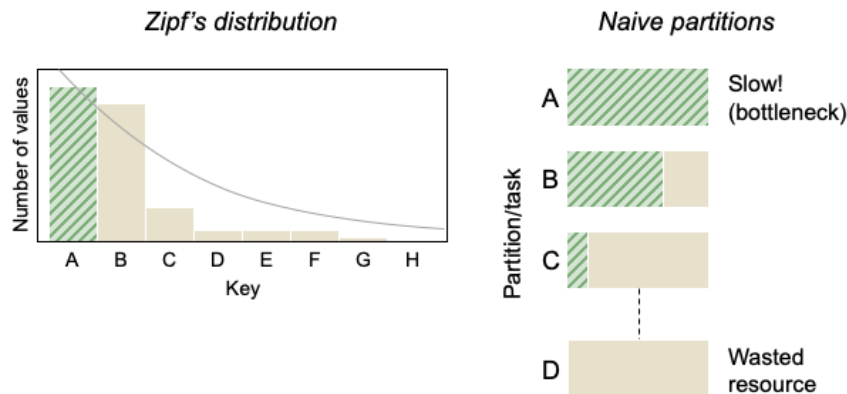
Spark provides some built-in functions for basic operations. For example, coalesce effectively collapses partitions on the same worker in order to avoid shuffling of the data when re-partitioning.

And repartition will cause a shuffle, but can be used to increase the level of parallelism when operating map and filter type operations.

repartitionAndSortWithinPartitions allows the programmer to specify both the partitioning key, as well as the key comparisons (sorting). We saw this last week when we implemented relative frequencies of co-occurring terms. Think about how you might use this to implement a total order sort.

# Custom Partitioning

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



Those are really nice, but sometimes you will need to perform some very low level partitioning because you're working with very large data and large key-skew. Key skew simply means that some keys have many more values than others - this is not an unusual scenario especially in certain domains like text mining, where we commonly deal with zipfian distributions. If you haven't heard of this term before, take a look at Zipf's Law: [https://en.wikipedia.org/wiki/Zipf%27s\\_law](https://en.wikipedia.org/wiki/Zipf%27s_law)

Custom partitioning is indeed the main reason one would use RDDs in a production setting. Custom partitioners are not available in the Structured APIs since they don't really have a logical counterpart. This low level implementation can have a significant impact on whether the job runs successfully.

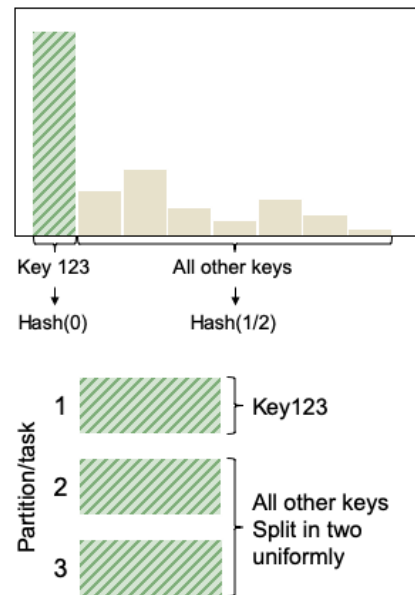
When faced with heavy key-skew as illustrated in this slide, you want to break the keys as much as possible to improve parallelism and prevent OutOfMemory errors during execution.

# 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]
```



Let's take a look at a simple example.

To perform custom partitioning you need to implement your own class that extends Partitioner. You should only do this when you have lots of domain knowledge about your problem space.

Let's say we have a dataset where one customer contains most of the data, and should be operated on alone, and all the other customers can be lumped together. This is obviously a gross oversimplification, but not the beyond the realm of imagination.

Our data is organized such that the key is the customer id.

We define a partition function where we assign a new partition key of 0 if the customer id is "123", and for all other customers, we randomly assign either a "1" or a "2".

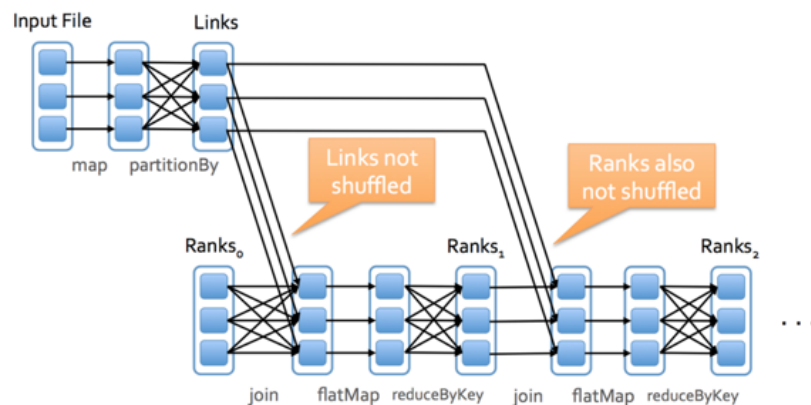
We can then use the partitionBy RDD function to specify the number of partitions, in this case 3, and our previously defined partition function to use for assigning our new custom partition keys.

The rest of the code calculates the number of keys in each partition, and returns this as a list to the driver.

This code and other examples, can be found in the github repository for Spark, The Definitive Guide book. <https://github.com/databricks/Spark-The-Definitive-Guide>

# A Case for RDDs

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



Another canonical example to motivate custom partitioning is PageRank. We won't get into the details today, but suffice it to say, that clever partitioning of the data helps us to improve performance by avoiding expensive shuffles. In a production setting (as opposed to a pedagogical one such as ours), if you're going to use custom partitioners, you should drop down to RDDs from the Structured APIs, apply your custom partitioner, and then convert back to a DataFrame or DataSet. This way, you get the best of both worlds, only dropping down to custom partitioning when you need to.

We'll see this again later in the course when we discuss graph algorithms. You will be asked to implement PageRank for homework, so remember to think about about partitioning when the time comes!

Controlling Partitions

---

# The End

That concludes our partitioning overview, next we'll talk about monitoring and debugging.

# Monitoring and Debugging

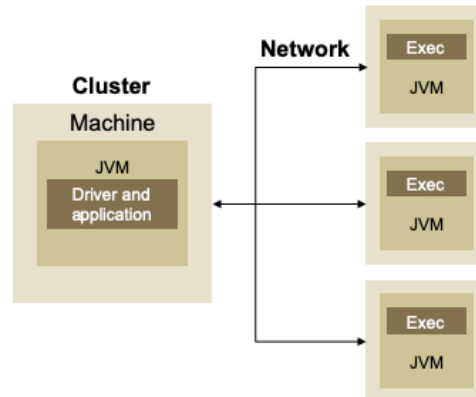
---

Chapter 18 Spark: The Definitive Guide



# 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*

At some point you'll need to monitor your Spark jobs to understand where the issues are occurring in them. In this section we'll review the different things that we can actually monitor and outline some of the options for doing so.

Let's review the components:

Spark applications and jobs:

The first thing you'll want to begin monitoring when either debugging or just understanding better how your application executes against the cluster is the Spark UI and Spark logs. These report information about the applications currently running.

JVM:

Spark runs the executors in individual Java Virtual Machines (JVMs). So the next level of detail would be to monitor the individual machines to better understand how your code is running. There are JVM utilities for reporting stack traces, creating heap-dumps, or reporting time-series statistics. Some of this information is provided in the Spark UI, but for very low-level debugging you'll want to use tools like *jstat*, *jstack*, *jmap*, *jconsole*. These are all JVM specific utilities.

OS/Machine

In addition to the JVM, you'll want to monitor the state of the OS and machines that

the JVMs are running on. Things like CPU, network, and I/O. Sometimes these are reported in cluster level tools, but there are also specific tools you can use like dstat, iostat, and iotop.

Cluster:

And of course, you can monitor the cluster on which your Spark application will run. This could be YARN, Mesos, or a standalone cluster. It's a good idea to have some sort of monitoring solution here since, if your cluster is not working, you should know pretty quickly about it.

## What To Monitor

---

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

So with that quick overview, let's talk about how we can go about monitoring and debugging our Spark application. There are two main things you'll want to monitor: the processes running your application, things like CPU usage, memory usage, etc.. And the query execution, so jobs and tasks.

Although processes are important, you'll probably want to start with debugging what's actually happening at the query level: queries, jobs, stages and tasks. There are two ways to get started here, Spark Logs, and the 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
```

One of the most detailed ways to monitor Spark is through its log files. These will help you to see exactly where jobs are failing or what is causing those failures. Unfortunately Python won't be able to directly integrate with Spark's Java based logging library, but.. You can use Python's logging module or even simple print statements to print to standard error. The logs will be printed to standard error when running in local mode, or saved to files by your cluster manager when running on a cluster. Sometimes you'll want to keep a record of past log files, or ship the log files off the machine running them so they are available even, and especially, after, a machine crashes.

# 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

2 Pages. Jump to 1. Show 100 Items in a page. Go

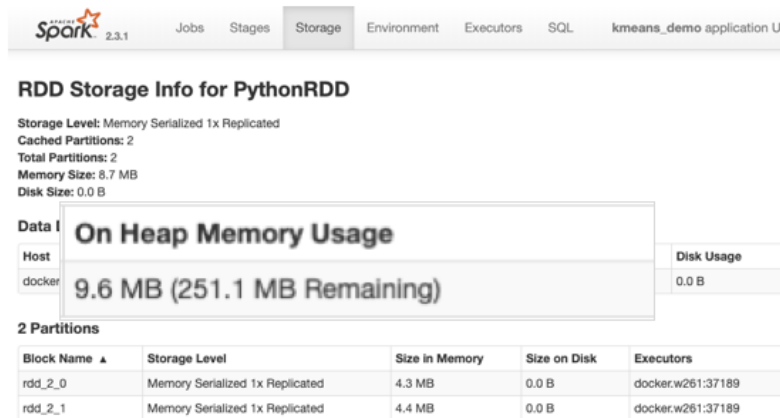
Task ID	Getting Result Time	Peak Execution Memory	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	0 ms	0.0 B	10.9 MB / 312139	0.1 s	24.9 MB / 688	
1	0 ms	0.0 B	11.0 MB / 312632	0.1 s	24.9 MB / 679	
2	0 ms	0.0 B	10.9 MB / 311604	0.1 s	24.8 MB / 688	
3	0 ms	0.0 B	10.9 MB / 311401	0.1 s	24.8 MB / 676	
4	0 ms	0.0 B	11.0 MB / 312740	0.1 s	24.9 MB / 691	
5	0 ms	0.0 B	10.9 MB / 312740	0.1 s	24.9 MB / 688	

The Spark UI provides a visual way to monitor applications while they are running as well as metrics about your Spark workload. Every Spark context running launches a web UI on port 4040.

For the most part, the tabs in the Spark UI should be self explanatory. Jobs tab refers to Spark Jobs, Stages refers to individual stages and their tasks. Look out for uneven distributions. Here we have an example where each task is handling about 11 MB of data. This is a nice even distribution. You could have a problem if for example, one task was trying to process 100MB, and another just 1MB. Then you might think about writing a custom partitioner to more evenly distribute your data among the tasks.

# The Spark UI

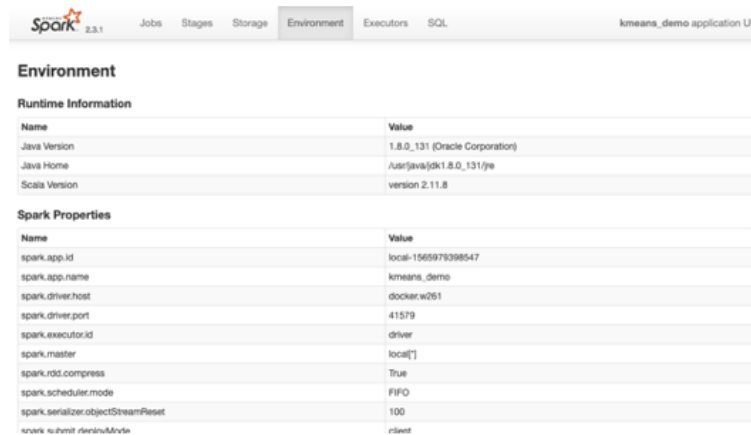
- Summary metrics for completed tasks—look out for uneven distributions!
- **Storage tab:** gives insight to garbage collection
- Environment tab: your Spark configuration



Storage includes information about the data which is currently cached. It gives you insight into garbage collection. If the amount remaining is very small, it could mean that objects are being held in memory and eventually you could run out. For example, keep an eye on your broadcast variables in iterative jobs! Imagine we are learning a KMeans model, and at every iteration, we resend the model file to all executors. After many iterations, we could end up with a lot of old model files taking up memory unnecessarily.

# 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 screenshot shows the Spark UI interface with the 'Environment' tab selected. The top navigation bar includes 'Jobs', 'Stages', 'Storage', 'Environment', 'Executors', and 'SQL'. The 'Environment' tab displays two sections: 'Runtime Information' and 'Spark Properties', each with a table of key-value pairs.

Runtime Information	
Name	Value
Java Version	1.8.0_131 (Oracle Corporation)
Java Home	/usr/java/jdk1.8.0_131/jre
Scala Version	version 2.11.8

Spark Properties	
Name	Value
spark.app.id	local-1565979398547
spark.app.name	kmeans_demo
spark.driver.host	docker.w261
spark.driver.port	41579
spark.executor.id	driver
spark.master	local[*]
spark.rdd.compress	True
spark.scheduler.mode	FIFO
spark.serializer.objectStreamReset	100
spark.submit.deployMode	client

The Environment tab contains relevant information about configuration and settings. SQL refers to the Structured APIs, and then the Executors tab provides detailed information about each executor running in the application.

These should be pretty self explanatory.

# Spark REST API

So you can build your own reporting tools!

<http://localhost:4040/api/v1>



Spark also allows you to access various metrics via a REST API so you can build your own reporting tools on top of Spark itself.



## Spark UI History Server

---

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

Normally, the Spark UI is only available while a SparkContext is running. So how can we get to it after an application crashes or finishes? That's where Spark's History Server comes in. You will first need to configure your application to store event logs in some location of your choice. You'll also need to set the `eventLog.enabled` property to true.

Then you can run the history server as a standalone application, and it will automatically reconstruct the web UI based on the logs.

Lots of other history server configurations are available, and you should check Spark documentation for specifics.

Controlling Partitions

---

# The End

That concludes our partitioning overview, next we'll talk about monitoring and debugging.

# Spark First Aid

---

Chapter 18 Spark: The Definitive Guide

# Debugging and Spark First Aid



Stage ID	Description	Submitted	Duration	Tasks Succeeded/Total	Input	Output	Shuffle (Spill)	Shuffle (Write)
82	groupByKey at org.apache.spark.sql.execution.shuffle.ShuffleExchangeExec\$GroupByKeyTask.run	2018/05/16 10:43:09	71 ms	0/2	4.2 KB	0.0 KB	0.0 KB	0.0 KB

Stage ID	Description	Duration	Tasks Succeeded/Total	Input	Output	Shuffle (Spill)	Shuffle (Write)
83	collect at org.apache.spark.sql.execution.ShuffleExchangeExec\$CollectTask.run	5 ms	0/2 (1 failed)				

OK, now that've identified how to monitor various operations, let's look at some common issues and how to address them.

Things like slow tasks, and outOfMemory errors.

# 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.

This issue can arise frequently, especially when you're just getting started with a fresh deployment or environment.

This mostly happens when your cluster or your application's resource demands are not configured properly. During the process of configuring your cluster, you might have configured something incorrectly, and now your driver is not able to talk to the executors. Perhaps you didn't specify the correct IP or ports. Or, maybe your application requested more resources than your cluster manager currently has free, and as a result, the driver will be waiting forever for executors to be launched.

Another common issue may be that you requested more memory per executor than the cluster manager has free to allocate.

Run a small simple application to see that everything works. Check the spark-submit memory configuration.

## 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...

Now that you know your cluster is set up correctly, and you've successfully ran applications, you are developing a new application, and the new code doesn't work.

After checking and confirming that the Spark UI environment tab shows the correct information, it's worth double checking your code.

Check for incorrect file paths, or typos.

Check for issues with libraries and class paths.

Try simplifying your application until you get a smaller version that reproduces the issue.

# 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

Errors during execution occur when you're already working on a cluster, or parts of your application run before you even encounter an error. This can be part of a scheduled job that runs at some interval, or part of some interactive exploration that seems to fail after some time.

First, check the stack trace. The stack trace will contain information about the components involved like the operator, or the job that was running and failed. Check your data format. Sometimes input data formats change over time. If you can't find the fault in the logs or the UI, try removing logic until you can isolate the problem.

## 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 tasks and stragglers – this issue is quite common when optimizing applications, and can occur either due to work not being evenly distributed across your machines (we talked earlier about key-skew) or due to one of your machines being slower than the others (for example, hardware problems).

One particularly common case is that you use a `groupByKey` operation and one of the keys just has a lot more data than others. Then you might look at the Spark UI for the shuffle sizes on each task.

Some potential treatments include:

Increasing the number of partitions

Doing some custom partitioning after getting a better idea of the data distribution

You can also try increasing the amount of memory allocated to your executors if that's possible – that's not always possible.

If you're using User Defined Functions, make sure they are not wasteful in object allocation or business logic.

Another thing you can try is turning on speculation – we'll talk more about that later.

This can help with faulty nodes, but does come at the cost of allocating additional resources.

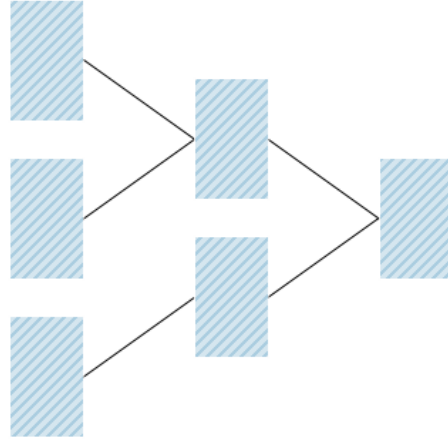
Stragglers can be one of the most difficult issues to debug, simply because there can



be so many causes. However, usually it's due to some kind of data skew which you can check for in the Spark UI relatively easily.

# 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



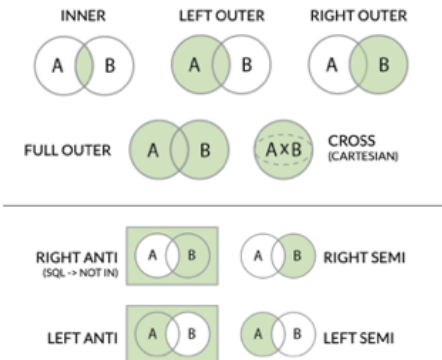
Slow aggregations are similar in symptoms and treatments to slow tasks. First try the techniques already mentioned, like increasing the number of partitions, for example.

Unfortunately this issue can't always be solved. Sometimes the nature of the data and the task at hand is just inherently slow. Not all aggregation functions are created equal. Try thinking through your logic to see if you can achieve the same results using different types of aggregations.

# 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!



Both joins and aggregations are shuffles, so they share some of the same general symptoms as well as treatments.

Many joins can be optimized to other types of joins.

Experimenting with different join orderings can really help if those joins filter out a large amount of data. Do those first.

Partitioning before joining can be very helpful in reducing movement across the cluster, especially if the same dataset will be used in multiple join operations. Just remember that this comes at the cost of an additional shuffle, so you need to experiment with the tradeoffs.

We'll talk more about broadcast joins in the structured API later on in the course, but just for completeness here, sometimes Spark can't properly plan for broadcast joins if it doesn't know about the statistics of the input dataframe or table. If you know that the tables you are joining are small, you can force a broadcast instead of relying on Spark to automatically use one.

We'll talk more about joins in general later in the course.

# 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

**Details for Stage 06 (Attempt 0)**

Total Time: 00:00:00.000  
Locality Level: Summary: Process Local 0  
Input Size: 0.0 MB  
Shuffle Write: 0.0 MB

**Summary Metrics for 2 Completed Tasks**

Metric	Min	90th percentile	Median	95th percentile	Max
Duration	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Shuffle Read	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Task Completion Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Get Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Result Transformation Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Input Transformation Memory	0.0 MB	0.0 MB	0.0 MB	0.0 MB	0.0 MB
Input Size - Records	0.0 MB	0.0 MB	0.0 MB	0.0 MB	0.0 MB
Shuffle Write Size - Records	0.0 MB	0.0 MB	0.0 MB	0.0 MB	0.0 MB

**Aggregated Metrics by Executor**

Index	ID	Locality Level	Exec ID	Total Duration	Shuffle Read	Result Transformation	Getting Result	Input Transformation	Input Size	Shuffle Write
0	1	PROCESS_LOCAL	driv	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 MB	0.0 MB	0.0 MB

Slow I/O

This can be difficult to diagnose with networked file systems

Turning on speculation can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it's just some transient issue in the first tasks. Speculation is a powerful tool and works well with consistent file systems. However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using.

Ensuring sufficient network connectivity can be important – your spark cluster may simply not have enough total network bandwidth to get to your storage system

And then for distributed files systems like HDFS running on the same nodes as Spark, makes sure Spark sees the same hostnames for nodes as the file system. This will enable Spark to do locality aware scheduling. See the locality column in the Spark UI.

It's OK if some of this is going over your head at the moment, as it's not really the focus of this course to delve too deeply into the engineering side of things. However, you should be aware of the potential issues at least at a high level.

# 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?

This is usually a pretty serious issue because it will crash your Spark application. It often happens due to collecting too much data back to the driver.

## Other Errors

---

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

Executor OutOfMemoryError:

Spark can often recover from these automatically. Some similar issues we talked about earlier can be relevant here. Things like garbage collection, inefficient UDFs, using an empty string to indicate a null value instead of an actual NULL.

If you're getting a lot of unexpected null values in your results, chances are your data format changed, but your business logic didn't. You can use accumulators to count records of certain types, or records that would have been skipped in try/except clauses.

No space left on disk is exactly what it sounds like. You may need to either allocate more disk space, or sometimes, if you have a lot of old log files, then these can be removed to free up some space. You can configure how long to keep logs. It's a not a perfect solution, but can be of some temporary help.

Serialization errors don't come up as often. Unless you're working with some very unique data types, this is not usually an issue.

## Conclusion

---

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

As with debugging any complex software, you should take a principled, step-by-step approach to debug issues. Add logging statements to figure out where your job is crashing and what type of data arrives at each stage, try to isolate the problem to the smallest piece of code possible, and work up from there. For data skew issues, which are unique to parallel computing, use Spark's UI to get a quick overview of how much work each task is doing.

In the next section we'll discuss performance tuning.

Monitoring and Debugging

---

# The End



# Performance Tuning

---

Chapter 19 Spark: The Definitive Guide

# 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

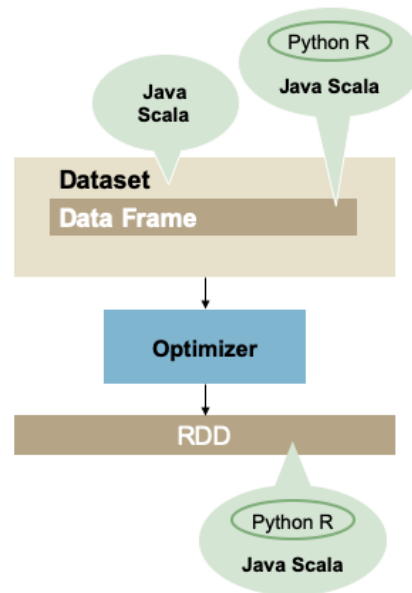
In the last section, we looked at some basic first-aid for your Spark application to ensure that your jobs run reliably. In this section we'll discuss some performance choices that can help your jobs to also run faster and more efficiently.

There are different parts of Spark jobs that you may want to optimize. This is a non-exhaustive list of some of these parts.

In addition, we can either do so Indirectly (through configuration settings), or directly through the design choices at the individual Spark job, stage, or task level.

## Indirect Performance Enhancements

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



Making good design choices at the configuration level that best suit your use case is the first step in making sure your Spark applications run in a stable and consistent manner. And that they remain stable and consistent over time and in the face of external changes or variations.

Let's start with choosing the programming language. There is no one single answer and a lot will depend on your use case. For example, if you want to perform some single-node machine learning after performing a large ETL job, you might consider writing your ETL in PySpark and then use python's massive ecosystem of machine learning libraries to run your ML on a single node. This will give you the best of both worlds, since as we know from earlier, Spark's Structured APIs are consistent across all languages. Things get a little trickier when you need to include custom transformations that cannot be created using the Structured APIs. These could be RDD transformations, or UDFs, and in this case, python may not be the ideal choice. However, you can still use python for the majority of your application code, and port your UDFs to Scala over time and as your application evolves.

Which brings us to the question of DataFrames vs SQL vs Datasets vs RDDs. This is a somewhat simpler question, since we know that DataFrames, SQL, and Datasets are equivalent in speed across all languages. If you need to write some transformations using RDDs, then using python will definitely incur a performance hit, mainly due to

the data serialization and deserialization that takes place. This can be very expensive to run on large datasets, as well as prone to decreasing stability.

## Indirect Performance Enhancements

---

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

The default serialization in Java can be quite slow. If you're working in Java or Scala, then Spark can use the Kryo library to serialize objects up to 10x faster. But you have to remember to register your classes. For this reason it is not the default. However, it is noteworthy that as of Spark 2.0.0 Kryo is used when shuffling RDDs with simple types, arrays of simple types, or string type.

In the python environment, Spark uses the pickle serialization library, and optionally you can specify the cloud pickle library which has support for certain complex data types the pickle does not.

## Indirect Performance Enhancements

---

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

This is probably more in the realm of data engineering, and even though it is not the focus of this class, it is noteworthy, that monitoring how the hardware is performing will be the most valuable approach toward optimizing your cluster configuration. Especially when it comes to running multiple applications (Spark or not) on a single cluster. Mainly it comes down to resource sharing and scheduling both at the application level and the cluster level. Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. We will not go into the details here, but you can dig into the Spark documentation to see how to enable this feature and set individual parameters.

## Indirect Performance Enhancements

---

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

Related to resource allocation, scheduling is another way of controlling resource utilization. Unfortunately there are no quick fixes here and research and experimentation is required. Cluster managers like YARN or Mesos, also provide some scheduling primitives that can be helpful when optimizing multiple Spark Applications.

## 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

We'll dive deeper into specific types of data at rest later in the course when we talk about data systems and pipelines. For now, it's important to appreciate that choosing your storage system, choosing your data format, and taking advantage of features such as data partitioning is essential to successful big data projects. There are many file formats available, from CSV, to binary blobs, to more sophisticated formats like Apache Parquet.

The most efficient file format you can generally choose is Apache Parquet. Parquet is column oriented and has certain characteristics that make it particularly amenable to parallelization. But whatever format you choose, you need to make sure that it is splittable, ie., different tasks can work on different parts of the file in parallel.

Other considerations when organizing data at rest:

Table partitioning - partitioning your files correctly allows spark to skip irrelevant data. For example, if users frequently filter by date or customer id, then using those fields as keys to partition your data will greatly reduce the amount of data that needs to be read for most queries.

Bucketing – The essence of bucketing is that it allows Spark to “pre-partition” data according to how joins or aggregations are likely to be performed. Bucketing generally works hand-in-hand with partitioning as a second way of physically splitting up your



data.

**Number of Files** – In addition to bucketing and partitioning, you'll want to consider the number of files and the size of the files that you're storing. If there are lots of small files, you're going to pay the price listing and fetching each of those individual files. The tradeoff can be summarized this way: Having lots of small files is going to make the scheduler work much harder to locate the data and launch all of the read tasks. This can increase the network and scheduling overhead of the job. Having fewer large files eases the pain off the scheduler but it will also make the tasks run longer. In general, the recommendation is to size your files so that they each contain at least a few tens of megabytes of data. One way of controlling data partitioning when you write your data is through a write option introduced in Spark 2.2. To control how many records go into each file, you can specify the `maxRecordsPerFile` option to the write operation.

**Data Locality** – has to do with scheduling tasks as close to the data as possible. HDFS provides this option. Generally this is the default configuration.

**Statistics collection** – In the DataFrame API, statistics are available on named tables. This is a fast evolving area of Spark, so you should check the documentation and JIRA tickets for latest updates.

## Indirect Performance Enhancements

---

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

Configuring Spark's external shuffle service can often increase performance, but it does come at the cost of complexity and maintenance, and you may decide that it's not worth it in your particular use case. Serialization has a big impact on shuffle performance, so always choose Kryo if working in JAVA.

## 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

The Spark documentation includes some great pointers on tuning garbage collection for RDD and UDF based applications, and here we'll paraphrase a few key points.

The first step in garbage collection tuning is to gather statistics on how frequently garbage collection occurs and the amount of time it takes. You will need to configure the `extraJavaOptions` parameter. Next time you run your job, you will see messages printed in the worker's logs each time a garbage collection occurs.

Garbage collection is a multistage process, and it is somewhat beyond the scope of our discussion, but seeing if garbage collection is run too often or if a full garbage collection is invoked multiple times before a task completes, tells you that there isn't enough memory available for executing tasks, so you may want to decrease the amount of memory Spark uses for caching. There is a lot more information online, and the effect of garbage collection tuning is highly dependent on the amount of memory available as well the application itself. You should definitely delve into this further if you think it's causing a problem for your application.

## Direct Performance Enhancements

---

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

Some of these enhancements we've already touched upon, though not in the context of performance tuning. Let's review each one and its impact on performance.

Increasing the number of partitions

Moving filters to the earliest part of the job. Sometimes predicates will be pushed down to the data layer, meaning you can avoid reading data which is irrelevant to your job to begin with.

Repartitioning often involves a trade-off between shuffling and optimizing the load for each task. So you may need to experiment with this one.

In general you want to avoid using UDFs. Now for the purposes of this course, we will be working mainly in the RDD API anyway, so this only applies later when you're working with the structured APIs in production. But there are some options such as Vectorized UDFs which use Pandas which are somewhat better performing than regular old UDFs

Caching is another one of those operations that incurs a cost, in that data must be serialized, deserialized, and stored. So if you're re-using the same RDD over and over, this cost is probably worth it, but if you only intent to use your RDD once, it will only slow you down. In addition, there are different storage levels which can be used for

caching, such as as memory disk, or heap, or combinations of each. Note that caching is a lazily evaluated operation, and nothing will be persisted until an action is called. By default, Spark caches data in memory, and spilling to disk as needed.

We talked about Joins, aggregations and broadcasts briefly earlier, I won't belabor the point here. We'll be talking in more detail about these operations later in the course.

## 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.

There are many different ways to optimize your Spark Applications, in general, the main things you want to think about are these ^

Performance Tuning

---

# The End

## K-Means at Scale

---



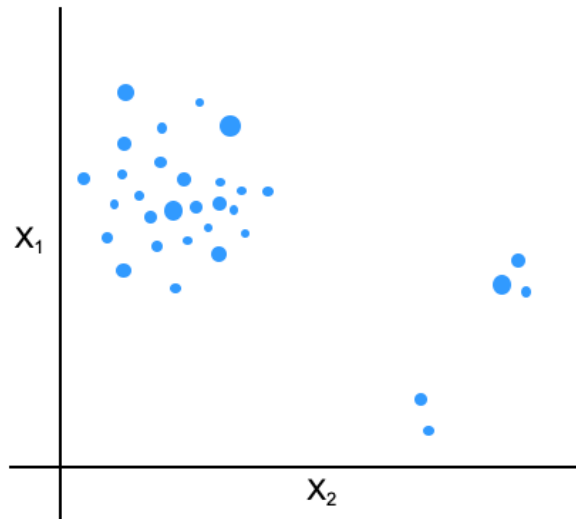
## A Case for Spark

---

- 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.

TODO: graphic

## Dataset



We'll use this highly contrived example dataset for the purposes of illustration. Even though our dataset is synthetic, it's not unusual for a real world dataset to resemble this distribution. Let's take a quick look at a couple of approaches.

## Version 1

---

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

Repeat steps 2 and 3 until convergence

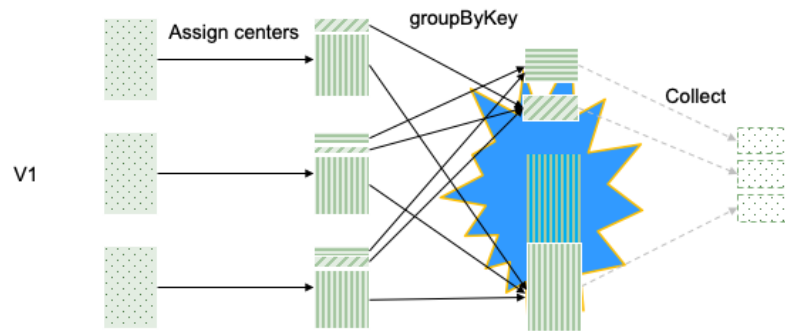
Intuitively, this seems like a good idea. It's a simple. We group all of the data by the centroid id, and then we recalculate all the averages for each centroid.

So what could go wrong here?

# 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.



All values end up in reduced tasks; the task could run out of memory if there is too much data in a single partition.

Because groupBy doesn't do any intermediate aggregation we could potentially send so much data to a single reduce task, and get the dreaded OutOfMemoryError.

But even in the case where we don't, we'd still be emitting more data over the network than necessary. Looking at the Spark UI, if we compare the shuffle size of this example with the next one, it's clear that this approach is not ideal.

## Version 2

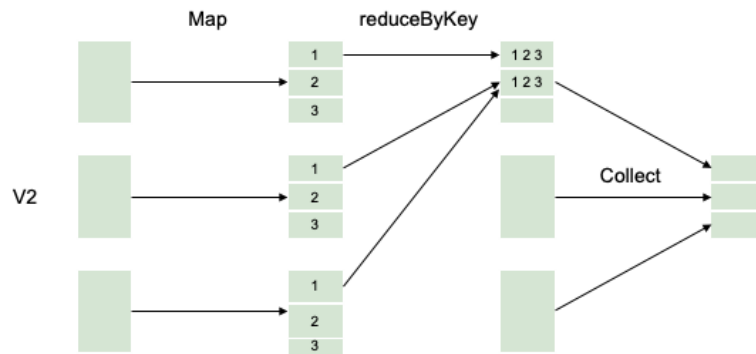
---

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

Repeat steps 2 and 3 until convergence

By using the `reduceByKey` function, we can take advantage of Spark's automatic use of combiners so that our reduce tasks only have to hold aggregated results! The trade-off is that our implementation is slightly more complicated. Now, instead of having a single function that calculates averages for a given key, we have to think about how we're going to combine the data points before the final average calculation. Remember from way back when we started, that averages are not commutative and associative, so we have to pass along the counts as well as the partial sums in our `reduceByKey` function.


## Version 2 Illustrated



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

Not only will our job actually finish, it will do so far more efficiently.

# Shuffle Write

 2.3.1

Jobs

Stages

Storage

Environment

Executors

SQL

**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

Our synthetic dataset consists of 100,000 records, with 3 clusters. Notice the shuffle write size in the groupByKey version is almost twice the size of the original data. Compare that to the reduceByKey, and it's not a stretch of the imagination to reason that at a much bigger scale, this could be the difference between our job succeeding and failing!

## Can We Do Better?

---

- Parquet ?
- Partitioning ?
- Caching ?

Some other things to consider:

Using the parquet format will make our data much smaller. However, this is a columnar format, and since we're accessing the data by row, it doesn't give us much in terms of access times.

Is there anything we can do to partition our data ahead of time? In this algorithm, the key is the centroid id, and this is something that changes at each iteration, so additional partitioning may not be of much help here.

Caching the RDD on the other hand, in an iterative job such as this one, makes a lot of sense, since we'll be accessing the same data set many times.



K-Means at Scale

---

**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!

Canopy clustering  
K-Means ||

Summary

---

**The End**