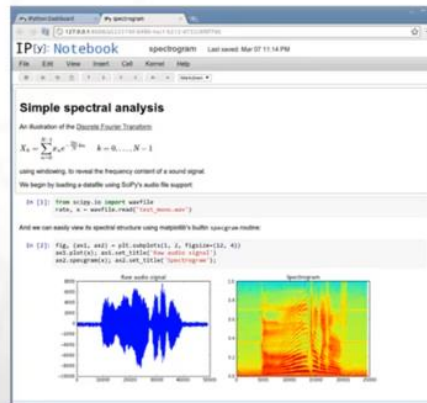


MODULE 1 – INTRODUCTION TO NOTEBOOKS

- Apache Incubator project started by NFLabs
- Interactive data analytics tool
 - Run code and create visualizations through web interface
- Plug in virtually any language backend
- Ships with Spark support
 - Scala with SparkContext
 - Markdown
 - Spark SQL
 - Shell



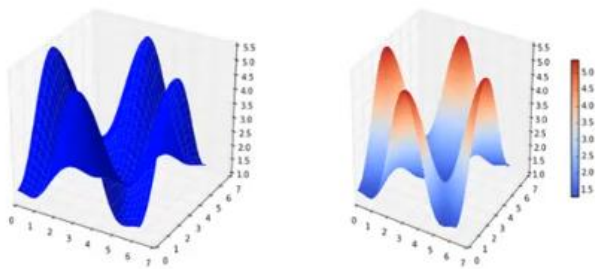
- Evolved from IPython
- Most mature
- Multiple languages
 - Julia, Ruby, R, Haskell, etc.
- pySpark Support
- Visualizations using pyplot
- <http://jupyter.org>



- Technology preview from IBM
- Interactive data platform built around Jupyter / IPython
- Preinstalled with Python, Scala, R
- Collaborate, search, and share notebooks
- Many tutorials and notebooks available
- Sign up at <https://datascientistworkbench.com>

```
p = ax.plot_surface(X, Y, Z, rstride=3, cstride=4, linewidth=0)

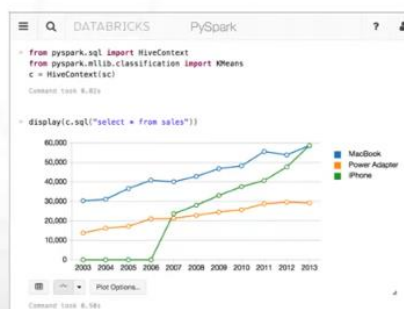
# surface plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                    cmap=plt.cm.coolwarm, linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```

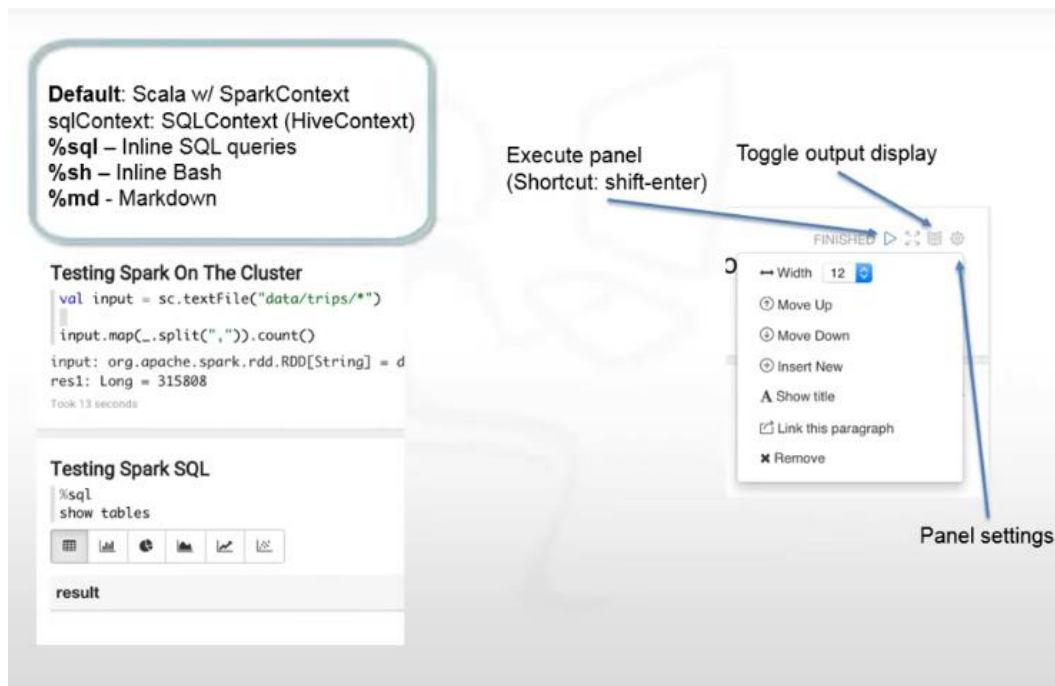


- Fork of Scala Notebook
- Full support for Spark
- Integrated Spark SQL support
- Dynamically inject JavaScript libraries (e.g. d3.js)
- <https://github.com/andypetrella/spark-notebook>



- Still in private beta
- Python, Scala, & SQL
- Currently, exclusively on AWS
- Monthly subscription
- Runs on your EC2 account
- Adding support for 3rd-party apps
- <https://databricks.com>





Lesson One: Introduction

This course was developed in collaboration with MetiStream and IBM analytics. This lesson is just a quick primer on data notebooks to get you ready for the lab exercises. After completing this lesson, you should be able to use Zeppelin in your projects and identify the various notebooks you can use with Spark. Apache Zeppelin is an interactive data analytics tool started by NFLabs. With Zeppelin you can run code and create visualizations through a web interface. You can plug in virtually any programming language or data processing backend. Zeppelin comes configured with Scala and Spark, as well as markdown, Spark SQL, and shell. You can visit the project homepage for more information. Jupyter is an evolution of IPython, a mature python-based notebook. IPython is still the Python kernel, but Jupyter also supports other languages including, Julia, Ruby, R, and Haskell. It supports pyspark, the spark api for python, and visualizations using the pyplot library. Visit jupyter.org for more information. The data scientist workbench is an interactive data platform in development from IBM. It's built around Jupyter/IPython notebooks. The workbench comes pre-installed with Python, Scala, and R. You can collaborate, search and share notebooks. There are many tutorials and notebooks available, ranging from introductory to using advanced libraries. You can register to preview the technology at datascientistworkbench.com. Another option is Spark notebook, a fork of Scala notebook centered on Spark development, with integrated spark SQL support. Spark notebook allows you to use JavaScript directly to create visualizations. One last notebook solution is databricks cloud. It's currently only available on Amazon Web Services and runs on your EC2 account. Here we see parts of the Zeppelin interface. You use special tags to identify which backend to use. The default (no tags) is Scala. A SparkContext is automatically instantiated with the variable "sc". There is also an SQLContext. The sql, sh, and md tags specify SQL, bash shell, and markdown, respectively. The play button, or shift-enter for short, executes the panel. You can toggle the output display and change the panel settings as well. You will become familiar with Zeppelin in the lab exercises. Having completed this lesson you should be able to use Zeppelin in your Spark projects and identify the various notebooks you can use with Spark. Proceed to exercise 1 and the next lesson.

>>Lab:

```
http://192.168.59.103:8080/#/
MINGW32/c/Users/IBM_ADMIN
initializing...
Virtual machine boot2docker-vm already exists
starting...
Waiting for VM and Docker daemon to start...
.
.
.
Started.
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\key.pem
Note
To connect the Docker client to the Docker daemon, please set:
export DOCKER_TLS_VERIFY=1
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH='C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm'
IP address of docker VM:
192.168.59.103
setting environment variables ...
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\key.pem
MINGW32/c/Users/IBM_ADMIN
setting environment variables ...
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
@37131dba0817/
Boot2Docker version 1.6.2, build master : 4534e65 - Wed May 13 21:24:28 UTC 2015
Docker version 1.6.2, build 7c8fca2
docker@boot2docker:~$ docker run -it --name bdu_spark2 -P -p 4040:4040 -p 4041:4041 -p 8080:8080 -p 8081:8081 bigdatauniversity/spark2:latest /etc/bootstrap.sh
-bash
starting namenode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-namenode-37131dba0817.out
Started Hadoop namenode: [ OK ]
starting datanode, logging to /var/log/hadoop-hdfs/hadoop-hdfs-datanode-37131dba0817.out
Started Hadoop datanode (hadoop-hdfs-datanode): [ OK ]
starting resourcemanager, logging to /var/log/hadoop-yarn/yarn-yarn-resourcemanager-37131dba0817.out
Started Hadoop resourcemanager: [ OK ]
starting nodemanager, logging to /var/log/hadoop-yarn/yarn-yarn-nodemanager-37131dba0817.out
Started Hadoop nodemanager: [ OK ]
starting org.apache.spark.deploy.history.HistoryServer, logging to /usr/local/spark-1.2.1-bin-hadoop2.4/sbin/../logs/spark-student-org.apache.spark.deploy.history.HistoryServer-1-37131dba0817.out
Zeppelin start [ OK ]
```

Spark II (Cognitive Class)



The first screenshot shows a Zeppelin notebook titled "Testing Spark On The Cluster" with a "FINISHED" status. The code defines an input RDD and maps it to count the number of commas in each string. The output shows a single value of 315808. The second screenshot shows a Zeppelin notebook titled "Testing Spark SQL" with a "FINISHED" status. It contains two sections: "Testing Spark SQL" which runs the command "show tables;" and "Testing Cassandra" which imports the Cassandra SQL context and runs a query to select all data from a table named "demo.clicks".

```
val input = sc.textFile("data/trips/*")
input.map(_.split(",")).count()
input: org.apache.spark.rdd.RDD[String] = data/trips/* MappedRDD[37] at textFile at <console>:25
res37: Long = 315808
took 1 seconds
```

```
%sql
show tables;

Testing Cassandra
import org.apache.spark.sql.cassandra._
val cqlContext = new CassandraSQLContext(sc)
cqlContext.sql("select * from demo.clicks").collect.foreach(println)
import org.apache.spark.sql.cassandra._
cqlContext: org.apache.spark.sql.cassandra.CassandraSQLContext = org.apache.spark.sql.cassandra.CassandraSQLContext@3d7
[1]
```

Hello and welcome to lab 1.

This lab you will be setting up your lab environment. The environment that you will be using

for the exercises will be the Zeppelin notebook

and the notebook comes from a docker image

that you will be able to pull from the docker repository.

If you are using a Windows operating system

like I am, you will need to download boot2docker. Go ahead and refer to the instructions listed

on the bigdatauniversity page on how to set up the boot2docker

for your Windows operating system. If you're using a Linux OS

you will not need to do that, but there are also other instructions

for you to get started. Once you have boot2docker installed,

go ahead and run the boot2docker start. This will start up the vm and the docker daemon.

as it is shown here. Note the IP address

of the docker VM. You will be using this to access the Zeppelin notebook

later. It should be the same IP address,

if not, just pay particular attention to what yours is. When the boot2docker

has loaded,

you will be brought to the terminal. This means you are now ready to continue on with the lab exercise.

Alternatively,

when you need to start up your container

you can SSH into the docker VM first, and then

execute a command to start up the container. Here

is my terminal

that I used to access the docker VM. I used putty.

The reason I'm using putty instead of the boot2docker terminal

that I have here is because putty allows me to copy and paste code

directly into its terminal. It's just a little quicker for me to use.

if you would like to use putty, refer to the instructions

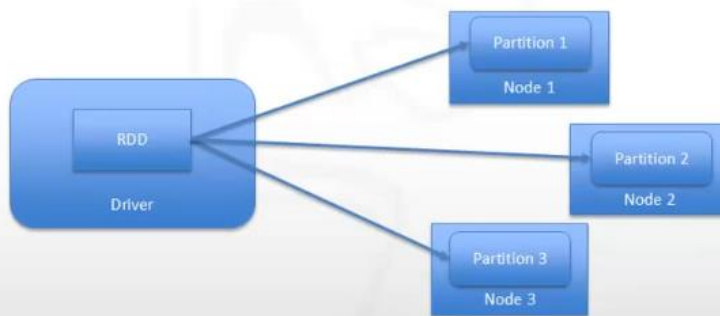
in the lab guide on how to set that up. The login

is docker and the password
is a public key
that you've imported with the configuration of putty.
so when you have done that, you don't need to provide a password. Once you're inside
the docker, you will execute the docker run command.
Again these instructions are
located on the bigdatauniversity page for this particular course.
You can copy and paste the command directly
and they will start up the container and load all the necessary services
for the lab environment.
Once everything has loaded, you'll notice at the very bottom
that Zeppelin Start is ok. This means your labs
are now ready to go. I'm going to move these out of the way
and bring the Zeppelin notebook
into focus. Remember, I said
the IP address of the VM will be used
and this is where you're using it 192.168.59.103
It should be the same for you,
if not, go back to the original boot2docker terminal
and find IP address. The port
to the Zeppelin service is 8080, so put this into
a web browser. I'm using Firefox here and you will be brought to this Zeppelin notebook.
Ok - so for the Zeppelin notebook, it is quiet easy to use, in fact
notice that there are individual lab links here. Another way to access the
links
or the labs rather, is to click on the notebook dropdown from above
and select the particular lab that you are interested in.
So we're going to click on lab 1
to bring up lab 1's book and
basically in lab 1, you are just to
verify that all the services are running properly, so there is a
Spark interpreter built in to this Zeppelin notebook
so that you can run Spark commands directly.
The first task but you should do is testing spark on the cluster.
It is just a basic loading of the text file and then
doing some map operation on it and then doing an action
to count number. This is just a test that Spark
runs. So go ahead and click on the play button
this will run the job.
First job, sometimes will take a while, if the server has not
started completely but evidently you should not get any
type of error messages, so that is what you're looking for -- a clean run.
This next button here will collapse and expand
the code section so you can see that I just
toggled it. You can hide the code you if you don't want to see it or you can expand it.
This next button
to that, the icon like a book will toggle the output display.
I'm hiding it
I'm showing in again. Because
this one is a basic command, you saw that it
created the RDD and then did the count on it. The file
contains this many records. That's actually all you need to do for this

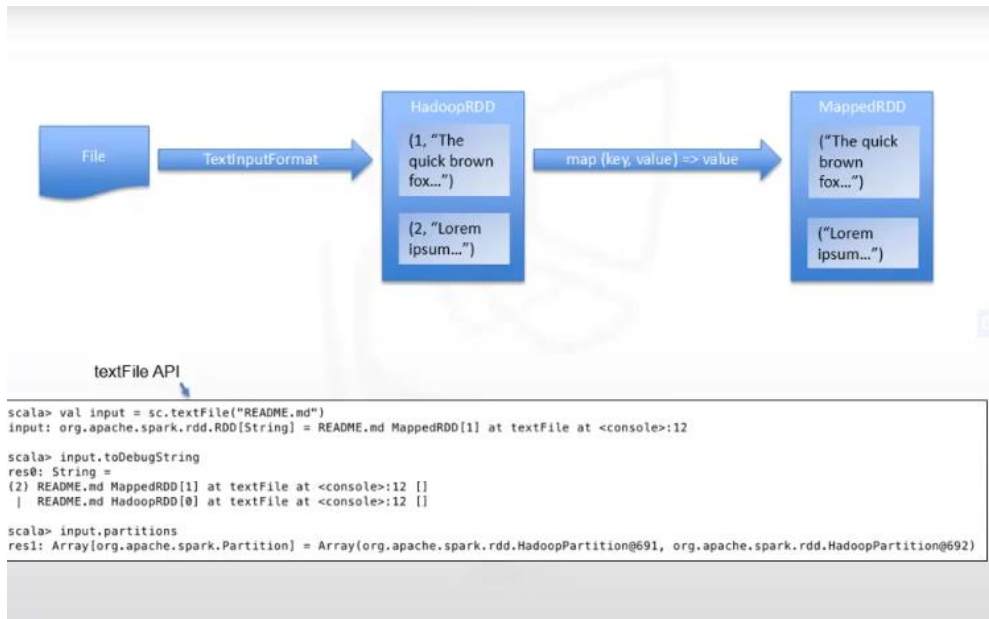
particular lab,
but just to point out that there are two
additional test that you would run if
you were doing the more advanced Spark activity
and these are not included as part of this particular exercise
so you can ignore these for now we don't have the services set up and running
but when we get to the more advanced concepts you will be using
these two.

MODULE 2 RDD ARCHITECTURE

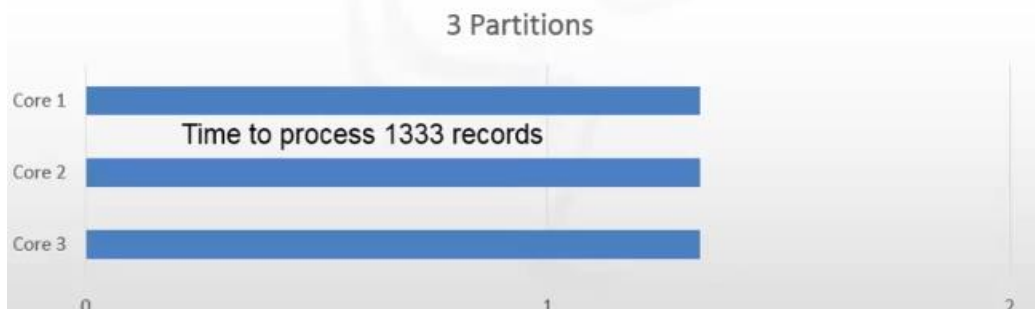
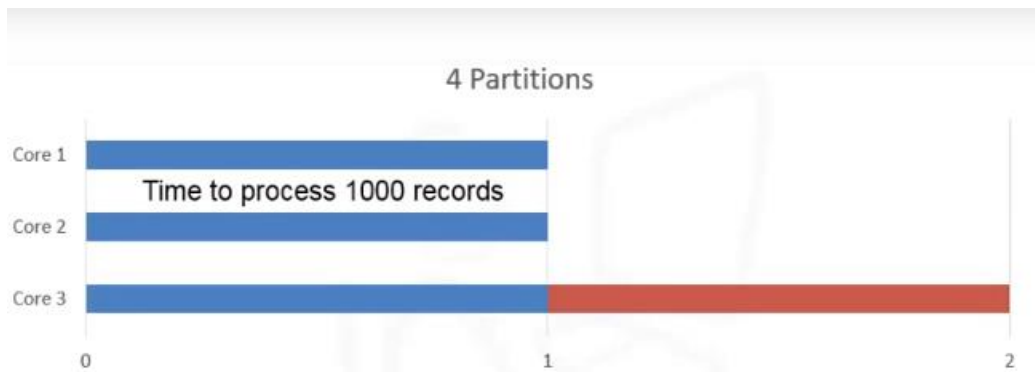
- **Resilient Distributed Dataset:** collection of elements **partitioned** across the nodes of the cluster that can be operated on in **parallel**
- Maintains a list of dependencies (previous RDDs in its graph)



- Uses Hadoop InputFormat APIs to input data
 - Support for many data stores
 - Not just HDFS
 - Local file systems
 - Cloud storage (Cloudant, AWS, Google, Azure, etc.)
 - HBase, Cassandra, MongoDB
 - Custom InputFormats
- InputFormat specifies splits and locality
 - Spark partitions correlate to HDFS splits
 - Observes data locality when possible
 - Optionally specify minimum partitioning



- Evenly distributed data
- Preferred locations
 - On systems like Hadoop and Cassandra, partitions should align with cores
- Number of CPU cores
- How long it takes to run a task
- Avoid **shuffling**
 - Movement of data between nodes
 - Very expensive
 - OOM errors
- It all depends on your data and the type of operations you will perform

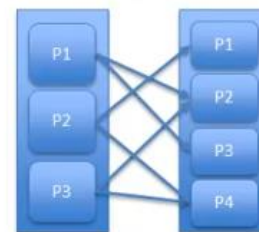


- Default partitioning is defined by input format
 - E.g., on Hadoop splits by HDFS cores
- Filter or map don't change partitioning

- **Repartition:** increase partitions
 - Rebalancing partitions after filter
 - Increase parallelism

```
repartition(numPartitions: Int)
```

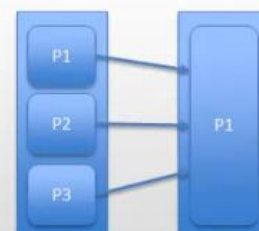
rdd.repartition(4)



- **Coalesce:** decrease partitions WITHOUT shuffle
 - Consolidate before outputting to HDFS/external

```
coalesce(numPartitions: Int,  
         shuffle: Boolean = false)
```

rdd.coalesce(1)



Lesson Two: RDD Architecture

This course was developed in collaboration with MetiStream and IBM Analytics.

After completing this lesson, you should be

able to understand how Spark generates RDDs, manage partitions to improve RDD performance,

understand what makes an RDD resilient, understand how RDDs are broken into jobs and stages,

and serialize tasks. By now you should be familiar with the Resilient

Distributed Dataset, the fundamental data abstraction in Spark. An RDD is made up of multiple partitions. Spark normally determines the number of partitions based on the

number

of CPUs in your cluster. Each partition has a sequence of records which tasks will execute on. The partitioning is what enables parallel execution of Spark jobs.

Recall that an RDD is part of a graph, sometimes referred to as a lineage, that traces its history. Spark uses standard Hadoop APIs for input.

Because of that, it's able to read from many different data stores in addition to

HDFS, including the local file system and cloud services like Cloudant, AWS, Google,

and Azure. Any InputFormat implementation can also be

used directly by the `hadoopFile` API for connecting with HBase, MongoDB, Cassandra, and more.

You can also implement your own custom InputFormats if necessary.

The partitions in the RDD map to the Hadoop splits as defined by the InputFormat.

This lets Spark take advantage of data locality when the Spark nodes are deployed on the Hadoop

Data Nodes. You can also specify a minimum partitioning

if necessary. Partitioning correlates to the parallelism

of tasks since each task executes on a single partition of data.

A key element of performance is balancing partitions with the number of cores on your cluster. Here is an example of using the built-in `textFile`

API. This is just a convenience function that calls the `hadoopFile` API, the same way you would read a text file in a MapReduce program. The `textFile` API simply calls the `map`

function

to discard the key and extract just the Text value from each record (line of text)

in the file. If we look at the graph of the full RDD you

can see that the root is a `HadoopRDD`, then a `MappedRDD`.

As we know, each RDD in a graph has a reference to its dependencies, up to the root RDD, which

depends on nothing. It's important to know how this affects partitioning. The root RDD

describes the original partitioning. These partitions are inherited by the child RDDs.

The transformations of each RDD, such as in the example above, are executed on each partition

of the parent RDD. However, we will soon see situations where a repartitioning occurs.

Partitioning can have a huge impact on performance in Spark. There are several factors to be consider that we will cover in this lesson. You want your data to be evenly distributed across partitions to leverage parallelism and reduce the chance of having a job that takes much longer than the others, causing a bottleneck.

On systems like Hadoop and Cassandra, it's crucial that their cores line up logically with partitions. You don't want a partition divided between two cores on separate

machines,

for example. The number of partitions should correlate

to the number of CPU cores in your cluster. Tasks that take a very long time to complete

might suggest a different partitioning is in order.

You always want to avoid shuffling, that is, the movement of data between partitions caused by certain operations. Shuffling is very expensive performance wise and can even lead to out-of-memory

errors. The big takeaway here is that there are many considerations and trade-offs when designing your cluster. It ultimately depends on your data and the type of operations you'll be doing on it.

Imagine that you have 4 partitions, with 1000 records each, on a cluster with three cores. Each task takes one hour to complete. Only three partitions can be run the task at once, so it would take two hours to complete all of them, and for half that time only one core is active. Now repartition to three partitions. Each task will take longer, since there are more records per partitions, but they can all run in parallel. The job finishes after only 80 minutes.

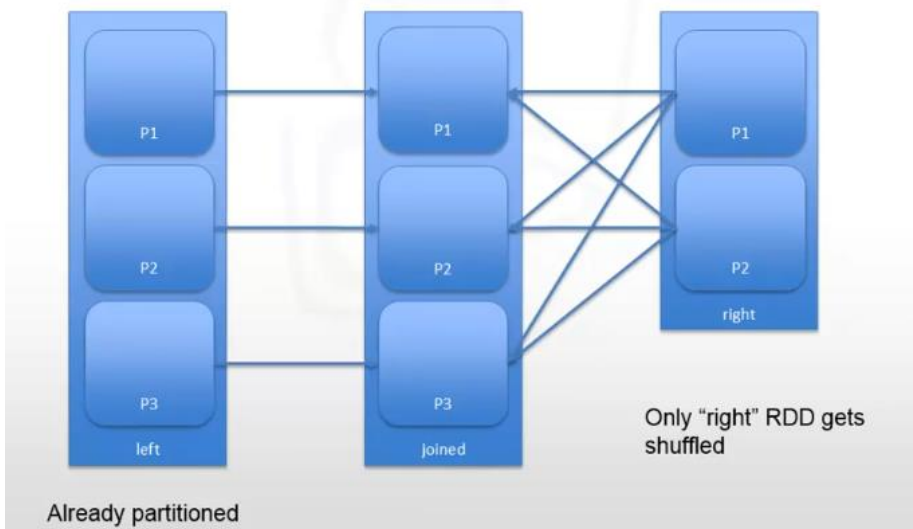
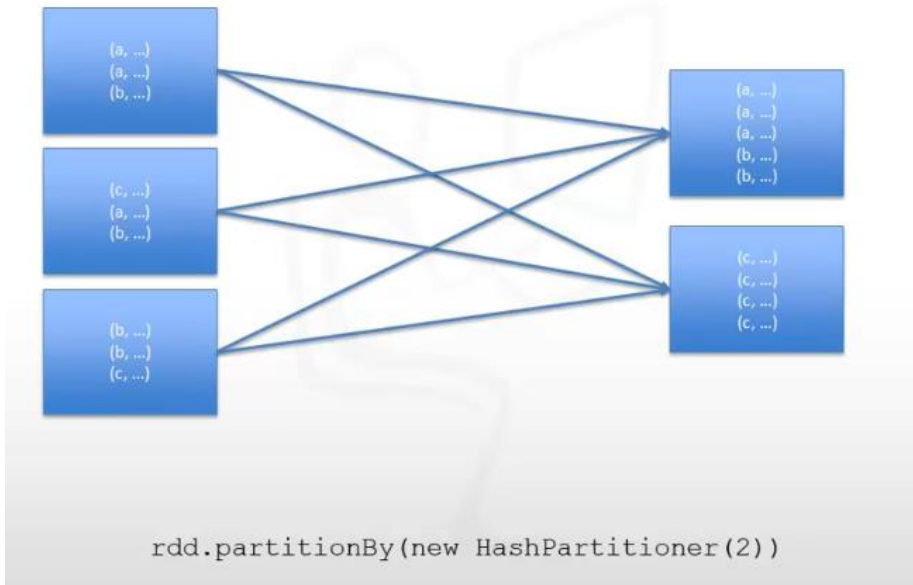
By default, partitions and the records within them are distributed based on the original storage system's InputFormat. For example, in Hadoop the partitions correspond with HDFS cores. RDD operations such as filter and map don't

alter partitioning. However in some cases you may want to force an increase or decrease in partitions. Repartition will cause a shuffle and redistribute partitions evenly. Repartition can be helpful to rebalance after filtering or reducing records and to increase parallelism if the input splits are too low for your cluster. Coalesce can decrease partitions WITHOUT incurring a shuffle. Coalesce is useful when you want to

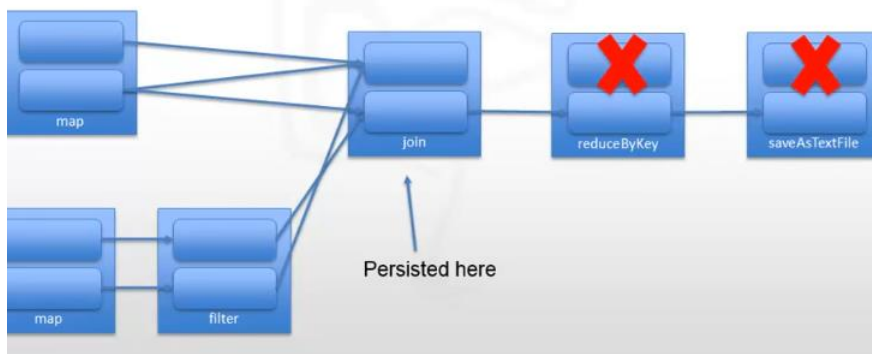
consolidate

partitions before outputting to HDFS or external systems, but you lose parallelism.

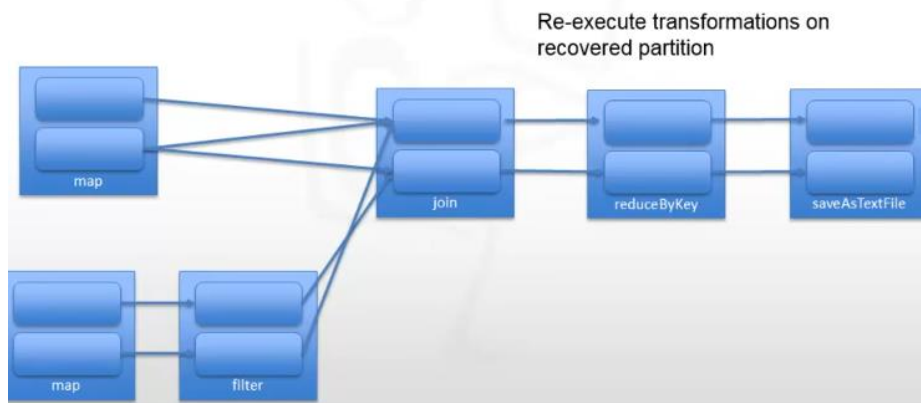
- Adding a key to an RDD does not change the partitioning
- Pairs with same key may not be in same partition (co-located)
- Co-location beneficial for many operations
- We can partition records by their keys
- **RangePartitioner**
- **HashPartitioner**
 - Ensures all pairs with the same key are co-located
 - $\text{Partition} = \text{Key} \% \text{numPartitions}$
- `partitionBy` causes a shuffle



- RDD lineage makes it resilient
- Re-compute results from root/persisted RDD



- RDD lineage makes it resilient
- Re-compute results from root/persisted RDD



- Job: sequence of transformations initiated by an action
 - collect, reduce, first, take, foreach, etc.
 - Job -> Stages -> Tasks
 - DAG Scheduler analyzes RDD to generate stages and tasks
 - **Stage boundaries** are defined by shuffle dependencies (AKA wide dependency)
 - Tasks are serialized and distributed to executors
-
- **Task**: operation defined in driver for an RDD
 - Closure (AKA lambda) is serialized and deployed to each executor
 - Must be **serializable**
 - Including any references
 - Be careful of large tasks
 - Serialization cost
 - Network I/O cost

When we key an RDD, either by a map or using keyBy, we're not changing the partitioning. All we've done is mapped each value to have a key. Those keys may or may not be trisie and we have no way of knowing since typically an RDD will not have a partitioner when it's first generated. However, for certain keyed operations or iterative processing it's beneficial to keep records with the same keys co-located. This lets our transformations run quickly within the same JVM when operating over the same keys. In order to do that we can use either a HashPartitioner or RangePartitioner. In the case of HashPartitioner we specify

the number of partitions and it will ensure that all keys with the same hash will be in the same partition. This does NOT mean each key will have its own partition. This is called consistent hashing, where the key hash modulo the number of partitions defines the partition

in which the record will be placed. Calling partitionBy will incur a shuffle but downstream operations will benefit from the co-located records.

Consider an RDD with 3 partitions and no partitioner. The keys are not co-located. If we plan on

doing keyed operations we'd be better off repartitioning so that all values for the

same key are in the same partition. Simply call `partitionby` with a hash partitioner with the desired number of partitions. A shuffle will occur, but further keyed operations will be more efficient. Joining RDDs that have no partitioner will cause each executor to shuffle all values with the same key to a single machine, for both RDDs. If you're repeatedly joining on the same RDD this is highly inefficient. The resulting RDD will use a `HashPartitioner` with the number of partitions equal to the largest RDD. If you need to join an RDD multiple times, a better option is to partition it. In this case, only the "right" RDD will get shuffled across the network to the corresponding partition from the "left" RDD. The best case scenario is that left and right RDDs are "co-partitioned" so they have similar keys and the same partitioner and same number of partitions. While a shuffle will still occur, if the partitions of different RDDs are on the same executor they won't cross the network. In general this will generate the least amount of network I/O and latency. Resilience RDD's describe a lineage of transformations that are lazily executed. As we've seen, each RDD depends on the parent. It describes some transformation to run over each partition of the parent RDD. Keeping this lineage allows us to reconstruct the results in the event of a failure. So if a partition is lost in an executor Spark only needs to recompute those lost partitions by walking up the lineage tree until it either reaches a root or persisted RDD. As an example, let's say we have two RDD's that are joined together. We then persist the RDD to memory. During the next operations, a `reduceByKey` and `saveAsTextFile`, a failure occurs and one of the partitions is lost. Spark is able to walk the RDD lineage until it finds the last known good state. Then it can recompute the lost partition. Note that it doesn't have to recompute every partition, just the one that was lost. Executing jobs

Spark actions can be thought of in three stages. A Job is a sequence of transformations

initiated

by an action, like `collect`, `reduce`, etc. Jobs are broken down into stages, which in turn consist of tasks. The scheduler analyzes the whole RDD lineage and determines how to break the job into stages and tasks.

Stage boundaries are defined by shuffle dependencies, i.e., when a join, repartition or other operation

that causes a shuffle occurs. Transformations that cause a shuffle are also called wide dependencies. Tasks are then serialized and distributed to the executors. Tasks are the operations we've defined in our Driver program for that RDD. The closures (AKA lambdas) we wrote in our code will get serialized then deployed to each executor with the partition on which that task must execute. The important thing to remember here is the fact that the tasks must be serializable since they have to get sent over the network to the worker nodes. This includes the tasks themselves as well as any references they make to objects and variables within the Driver. You have to be careful that your tasks aren't too large, for instance with large local variables (such as collections or maps). There are multiple

costs associated with large tasks. The first is serialization and de-serialization cost. Also there is the network I/O for sending those tasks each time that transformation is executed. Finally large tasks may delay task launch times.

Part 3

HOW TO TROUBLESHOOT "TASK NOT SERIALIZABLE"

```
class MyHelper {
  def doSomething(input: String): String = input
}
```

```
val helper = new MyHelper()
```

```
val output =
input1.map(helper.doSomething(_))
```

Does not extend
serializable

Even if we could modify
it, costly to serialize and
send entire object

```
class MyHelper extends Serializable {
  val inner = new ExternalClass()

  def doSomething(input: String): String = inner.doStuff(input)
}
```

```
val helper = new MyHelper()
```

```
val output =
input1.map(helper.doSomething(_))
```

```
class MyOtherHelper extends Serializable {
  def doSomething(input: Int) = input
}
```

```
object MySparkApp {
  val helper1 = new MyOtherHelper
```

```
def main(args: Array[String]) {
  val sc = new SparkContext(new SparkConf())
  val input1 = sc.parallelize(1 to 1000)
```

```
val localHelper = helper1
val output = input1.map(localHelper.doSomething(_))
```

```
output.collect()
```

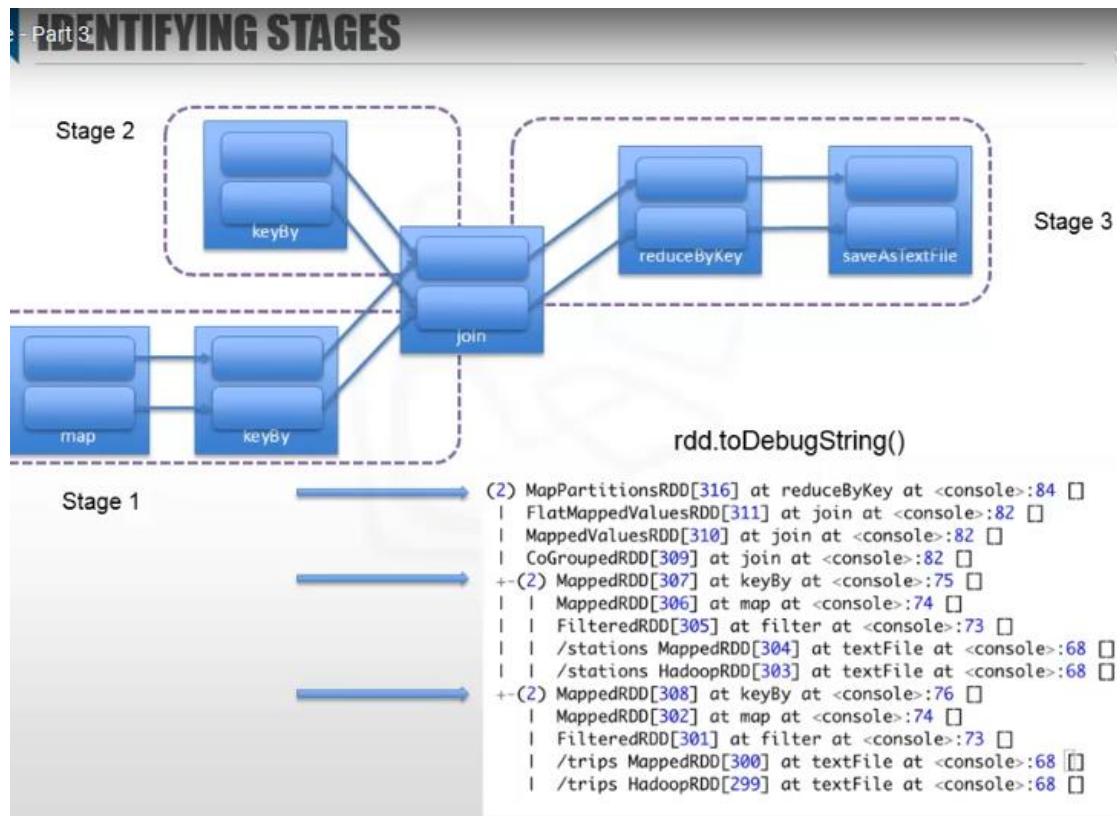
```
}
}
```

Member variable of
MySparkApp

Spark tries to serialize
MySparkApp

Serializing would send
entire app to every
worker

Solution: local
reference to helper1



Part 3 HANDLING STRAGGLERS AND FAILURES

- Speculative execution
 - slow running tasks in stage are re-launched
 - `spark.speculation = true` (default=false)
- “Slow” is defined by `spark.speculation.multiplier` (default=1.5)
 - How many times slower a task is than the median
- Task failures due to memory issues, hardware, network, etc.
 - `spark.task.maxFailures` (default = 4)
- May want to fail fast vs. running long task multiple times
- Be aware of side-effects
 - e.g., output to external system
 - If task fails and re-runs it might output same data again
 - Need to write custom code to be idempotent

CONCURRENT JOBS

- Default First-In-First-Out (FIFO) Scheduler
 - Fine for single-user apps
 - Each job gets as many resources it needs
 - Jobs from multiple threads can run in parallel if resources allow
 - But one large task will back things up
- Fair Scheduler is more appropriate for multi-user
 - `spark.scheduler.mode = FAIR` (default FIFO)
 - Resources allocated equally across jobs
 - Can define pools with weights/minimum share of resources
 - Pools can have their own scheduler (default FIFO)
 - Useful for priority queues
 - Zeppelin gives each user a pool

Part 3

USING FAIR SCHEDULER POOLS

```

<?xml version="1.0"?>
<allocations>
  <pool name="default">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="user1">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>4</minShare>
  </pool>
  <pool name="user2">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>4</minShare>
  </pool>
</allocations>

```

```

conf.set("spark.scheduler.allocation.file", "/path/to/schedulerpool.xml")

sc.setLocalProperty("spark.scheduler.pool", "user1")

dataset.foreachPartition(part => {

})

sc.setLocalProperty("spark.scheduler.pool", null) // sets back to default

```

Some common errors you may get, especially as you create more complex applications using 3rd-party libraries, are “task not serializable” errors. These typically crop up on complex tasks referencing member variables of the class defining the closures.

In this example we have a very basic helper class that doesn’t really do much. We reference it in a map operation. If we try to run an action on the output RDD we’d get an error. The problem is that “MyHelper” doesn’t implement Serializable. So if we extend our class with Serializable and try to run it again it will work properly.

What if “MyHelper” in turn is also referencing another 3rd party library as a member variable

which is not serializable? We may not want (or be able to) change the code to make it serializable. Also, because of the cost of sending large objects as part of our it would be better not to serialize the whole object graph.

What other option do we have? If we mark the member variable transient and lazy, it will not get serialized but will still be instantiated locally within each task and our code will run as expected, without incurring the cost of serializing a large task or lots of code changes. Here we have another example, this time of a full Scala app. We have a local class “MyOtherHelper” used in a map operation. In its current form

this example will fail, even though the helper class is serializable. Why?

Because “helper1” is a member variable of MySparkApp, and so Spark tries to serialize that instance as well.

We just could make MySparkApp serializable, but is it really a good idea to ship our whole app to every worker? A better option is to make a local reference to “helper1”. This way we’re keeping our task as slim as possible and minimizing serialization issues over large object graphs. As we discussed earlier, stages are defined by shuffle dependencies. Here we have a graph of 2 RDDs being joined, then a reduceByKey, and finally a saveAsTextFile. Here is the RDD lineage from calling to debug

string. The debug string clearly highlights the stage boundaries. Stage one and two are the original RDDs being filtered, mapped and keyed in preparation for the join.

The join requires a shuffle, so there is a stage boundary there. The third stage is the reduce and save after the join. Spark has a feature called speculative execution for handling slow tasks. As stages are running slow tasks will be re-launched as necessary. Speculative execution is disabled by default.

We can enable it by setting the spark.speculation parameter to true.

“Slow” is a configurable value set by spark.speculation.multiplier. This defines how many times slower a task is than the median to be considered for speculation.

We also need a way to handle failing tasks. Tasks could fail either due to memory issues, underlying hardware issues, networking problems, etc.

By default Spark will retry a task 3 times before failing a stage, which can be changed with the spark.task.maxFailures parameter as needed.

For example you may want to fail fast versus running a long task multiple times to find out there’s an issue on your cluster. An important issue to be aware of is any operation that has side-effects, such as outputting data in a for each call.

If our task fails and gets re-run it may very well try to output that data again.

If you’re using custom code to write out to another system be aware that you may need to ensure the operation is idempotent. Most of what we’ve looked at so far has been a single-user working in the Spark shell or a single app being submitted working on a single task. By default, Spark uses a basic FIFO scheduler. It works well for single-user apps where each job gets as many resources it needs.

The SparkContext is fully thread-safe so within a Spark app we can actually submit jobs from multiple threads. If multiple jobs are submitted concurrently they could run in parallel if the first job doesn’t need all the cluster resources.

However, if that’s not the case you may have a single large job that backs up all other requests until it’s finished. For multi-user environments such as Zeppelin, you’ll want to use the Fair scheduler, which can be set with the spark.scheduler.mode parameter.

By default the fair scheduler divvies up the cluster resources in a round-robin fashion,

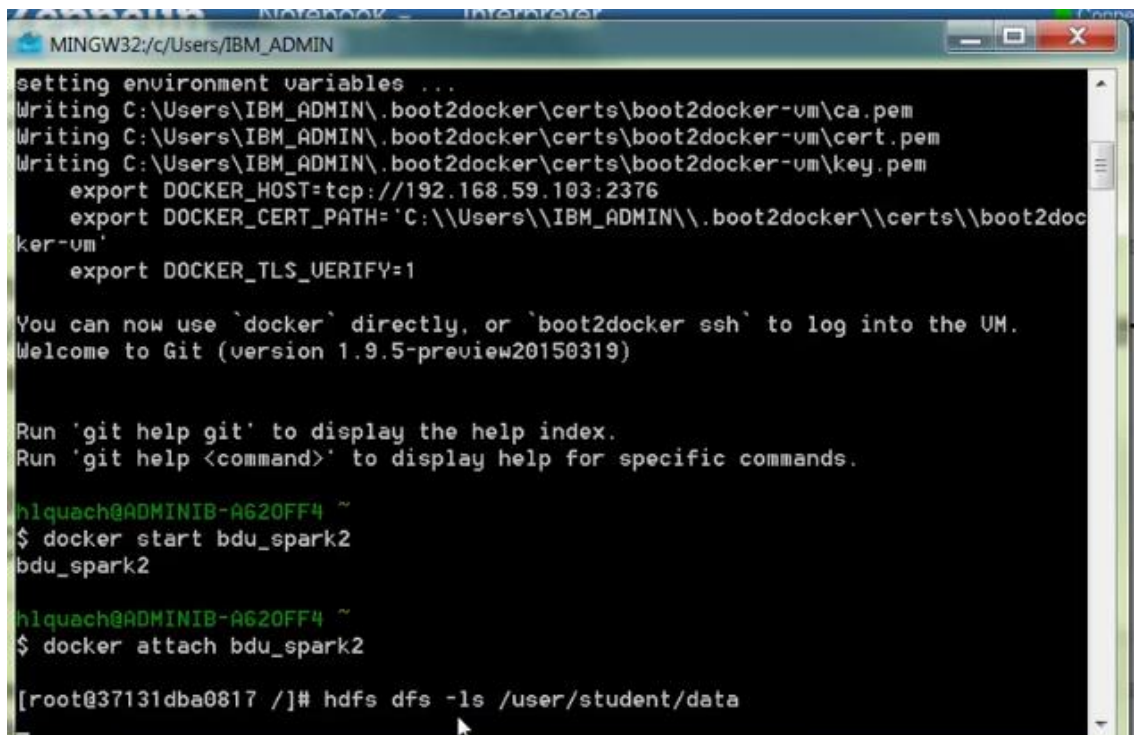
giving equal shares of resources. So even if you have a very large job it won't stop a small, short, job from starting and running. We can also define pools, that have custom attributes such as weights and minimum shares of the the cluster resources. Pools can have their own scheduler. Pools are useful for defining priority queues (like for a certain user or group) or by giving each user a pool which is internally a just a FIFO queue. This is the model Zeppelin uses to allow multiple users to work on the Spark cluster at the same time. Configuring custom Fair Scheduler pools involves first defining an XML file with the pools and their config properties. Then in your Spark app, or config settings, you need to set the `spark.scheduler.allocation.file` setting, pointing to that XML file. Submitting a job for a specific pool is done by setting a local property on the SparkContext, `spark.scheduler.pool`. This is a thread local property so if you spawn multiple threads on your driver each will have to set this value. To clear the value, just set it to null.

Having completed this lesson, you should be able to understand how Spark generates RDDs, manage partitions to improve RDD performance, understand what makes an RDD resilient, understand

how RDDs are broken into jobs and stages, and serialize tasks.

Proceed to exercise 2 and the next lesson.

>>Lab:



```
MINGW32/c/Users/IBM_ADMIN
setting environment variables ...
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\ca.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\cert.pem
Writing C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm\key.pem
export DOCKER_HOST=tcp://192.168.59.103:2376
export DOCKER_CERT_PATH='C:\Users\IBM_ADMIN\.boot2docker\certs\boot2docker-vm'
export DOCKER_TLS_VERIFY=1

You can now use `docker` directly, or `boot2docker ssh` to log into the VM.
Welcome to Git (version 1.9.5-preview20150319)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

hliquach@ADMINIB-A620FF4 ~
$ docker start bdu_spark2
bdu_spark2

hliquach@ADMINIB-A620FF4 ~
$ docker attach bdu_spark2

[root@37131dba0817 /]# hdfs dfs -ls /user/student/data
```



```

MINGW32/c/Users/IBM_ADMIN

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

hlquach@ADMINIB-A620FF4 ~
$ docker start bdu_spark2
bdu_spark2

hlquach@ADMINIB-A620FF4 ~
$ docker attach bdu_spark2

[root@37131dba0817 /]# hdfs dfs -ls /user/student/data
Found 5 items
-rw-r--r--  1 student supergroup      8196 2015-04-15 16:38 /user/student/data
/.DS_Store
drwxr-xr-x  - student supergroup      0 2015-04-15 16:38 /user/student/data
/sql
drwxr-xr-x  - student supergroup      0 2015-04-15 16:38 /user/student/data
/stations
drwxr-xr-x  - student supergroup      0 2015-04-15 16:38 /user/student/data
/trips
drwxr-xr-x  - student supergroup      0 2015-04-15 16:38 /user/student/data
/weather
[root@37131dba0817 /]# hdfs dfs -ls /user/student/data/trips

```

Create 2 New RDDs By Joining Trips With Stations By Start Terminal And End Terminal

ABORT ▶ ⌂ □ ⚙

```

import org.apache.spark._

val input1 = sc.textFile("data/trips/*")

val header1 = input1.first // to skip the header row

val trips = input1.
  filter(_ != header1).
  map(_.split(","))

val input2 = sc.textFile("data/stations/*")

val header2 = input2.first // to skip the header row

val stations = input2.
  filter(_ != header2).
  map(_.split(","))

/*
The id field for stations is index 0, the start terminal for trips is index 4, the end terminal is index 7.
*/

val startTrips = ???
val endTrips = ???

```

Hello, and welcome to lab 2.

This lab you will be joining RDDs

using partitioning, but before we go further

I wanted to show you what you would need to do if you

did have to restart your machine or ended up closing the docker image.

As usual you will have to start up boot2docker again if

you terminated it before.

And then you will use two commands to get back into the container.

First to start up the container if it hasn't been started. In this case because I

was in fact,

still using the same container and it hasn't stopped so,
it started up really quickly and they just need to attach to it.
Just go ahead and hit Enter again,
it'll bring you right into the docker container, and because everything was already up
the Zeppelin should still be up. I'll just minimize
the window here.

Now you're ready to begin lab 2. So in this lab, you will be
using two datasets, bike datasets.
and these bike datasets have already been loaded into the docker container
so you can check it out on the Hadoop filesystem.

So the first thing you want to do, is to go to the lab exercise guide
and I will show you the steps you will need to use in order
to take a look at this files. You can use
the boot2docker window or if you ssh into it, you can use that one.
it's up to you which one you wanted to choose. So inside this
container, you will go ahead and type in the command to
view the CSV files. So it's just `hdfs dfs`
and then do a listing under the user
student data directory
okay you've noticed several folders in here and these are all the data
well you won't be using all of the data
for this particular lab exercise. You'll be using some them.

So let's take a look at the CSV files. go ahead and do `HDFS dfs -ls`
right under `user/student/data`
and then the first CSV file will be located under `trips`, so if you want to take a look
at that, you can.

It's just a single CSV file
and the other one is located under `stations` but you can check that one out
yourself

let me move this out of the way
so now we're in the lab 2 notebook.

The first task that you will be creating here.
is and joining these two RDDs
with the stations by start terminal and the end terminal

It looks a little bit funky because I made my window smaller for the recording.
so you can maximize your screen, the text will show up a little better but that's
not important

so little bit about this data set. This data is from the San Francisco Bay Area
bike share program it is a public dataset
that the contains the trips and bike stations over roughly a 2-years period
and these are simple CSV file but they're enough to show you how
Spark will function with partitioning when you happen to join the two RDDs.

Alright, so
let's go over this code quickly. None of this should be
new to you, you should have seen this before. You have the
`input1` to create the RDD of the text file and in this case, we're just creating
a text file
from all the files within that `trips` directory, but you saw that it was only
one
and in our case that's true we only have one file
for the trips data. We're going to create the
another one to extract the header from it so that's all the header columns with the

header names.

We don't need those names within our data and then now you create your trips

RDD

You are going to filter out the header, and then you are going to split them up because

it is a CSV file, split all the words up by

the commas. You are essentially going to do the same thing for the stations

file so you have a trip state and now this is the station's data

and then you also going to split out the

split up the file and filter out the header as well and then now on this task

is what are the two operations needed

to join the station to the

trips by their station ID and then here's a hint

for the station index, so you don't have to look in the

file but we're telling you here the index 0 for stations is the station ID

for the trips it is at index 4

the start trip is index 4, for the end terminal is

is index 7, so what would you need to do here

to join these two RDDs?

Go back to start of transcript.

[Download video](#)

[Download transcript](#)

.txt

MODULE 3 OPTIMIZING TRANSFORMATIONS AND ACTIONS

Transformations and Actions - Part 1

ADVANCED RDD OPERATIONS

- Statistical operations on numerical RDDs
 - histogram, mean, stdev, sum, variance, max, min
 - stats() returns all of the statistic values
- mapPartitions, mapPartitionsWithIndex
 - Map by partition (many -> many) instead of single value/pair (one -> one)
 - Useful when map use function has a high overhead cost
 - E.g., connect to a database once per partition instead of for each record
- foreachPartition
 - Use for batching operations
- Approximate calculations
 - Get approximate results for very large data sets

```
def countApproxDistinct(relativeSD: Double = 0.05): Long
```

ADVANCED RDD OPERATIONS – CONTINUED

- `fold(zeroValue)((acc, value) => acc)`
 - Similar to `reduce`, but has Initial “zeroed” accumulator
 - Functions uses the accumulator and RDD element to update accumulator
- `aggregate(zeroValue)(accumulate, combine)`
 - Perform complex aggregations
 - Accumulate by partition then combine results
- `countByValue`

```
val text = sc.textFile("README.md")  
  
val counts = text.flatMap(_.split(" ")).map((_, 1)).reduceByKey(_ + _).collectAsMap()  
  
val counts = text.flatMap(_.split(" ")).countByValue()
```

Example: word count with `countByValue()`

OPERATIONS ON KEY/VALUE RDDS

- `reduceByKey`
 - Reduce over all values of a key
 - Combine results for each key in partition
- `countByKey`
 - Calls `reduce (_ + _)`
 - Collects results from every partition in a map **sent back to driver**
 - Mostly for testing/development convenience
- `aggregateByKey`
 - Aggregate over each key
 - Combine for each key of each partition
- `foldByKey`

OPERATIONS ON KEY/VALUE RDDS – CONTINUED

- `groupByKey`
 - Group all values by key from all partitions into memory
 - Potential cause of OOM errors
 - Intuitive, but **avoid** using when possible
- `Lookup`
 - Return all values for specified key
- `mapValues`
 - Apply a map function to each value without changing key
 - Spark optimizer knows partitions are still good after
- `repartitionAndSortWithinPartitions`
 - More efficient than calling `repartition` then `sortByKey`

Lesson 3: Optimizing Transformations and Actions

This course was developed in collaboration

with MetiStream and IBM Analytics. After completing this lesson you should be able to use advanced RDD operations, identify what operations cause shuffling, understand how to avoid shuffling when possible, and group, combine, and reduce key value pairs.

We'll start by going over some of the more advanced RDD operations available.

Numeric RDDs have several statistical operations that can be computed such as standard deviation,

sum, mean, max, min, etc. You can compute a specific operation or call `stats()` to return an object with access to all the values. `mapPartitions` is like regular `map`, except the function is at the partition level. Thus each set of values in a partition is mapped to zero or more values. One case when these operations are particularly useful is when your map function has a high overhead cost per record. For example, if you need to connect to a database, you could do it once per partition instead of for each individual record.

`mapPartitionsWithIndex`

simply adds the index of the partition as a parameter

`foreachPartition` - run an action that iterates over each partition. You want to use this for batching operations such as submitting to a web service or pushing to external data sources. You'd want to do this instead of calling `foreach` which iterates over each individual record. There are a number of operations that can

be used on extremely large data sets to get approximate values. For example,

`countApproxDistinct`

will approximate the number of distinct values to within the given standard deviation.

`Fold` is a special case of `reduce`. You pass it an initial zero accumulator and a function.

The function is a sequential or comparative operation between the RDD element and the accumulator, returning the final accumulator. `Aggregate` is similar to `fold`, but takes two functions. The first runs on each partition. The second combines the results from each

partition

accumulator. `countByKey` is just a convenience method

that counts the number of occurrences of each value and collects them in a map. Internally it's mapping to a pair RDD and calling countByKey. We can see how it works by revisiting our word count example. The second and third lines are identical operations.

Some operations can only be performed on Key-Value RDDs.

Reduce by key runs a reduce over all values of a key, then combines the results for each key in each partition. Count by key is simply calling reduceByKey, but the difference is that it collects the results in a map which is sent back to the driver. As such you need to be careful with this action - it's mostly useful as a simple way to examine your data and not something you want to use in production.

Aggregate and fold by key are analogous to their generic counterparts, except that they run per key. We will see an example shortly. Group by key groups all values by key, from ALL partitions, into memory. This action should trigger warning bells any time you see it in a Spark application. Recall that one of the worst culprits of poor performance in Spark is shuffling. groupByKey shuffles everything. It can even cause out-of-memory errors with

large datasets. avoid using whenever possible. Lookup returns all values for the specified key. MapValues applies a map function to each value

without changing their keys. When you use this instead of regular map, Spark knows that any previous partitioning is still valid, and so repartitioning isn't necessary.

Finally, repartition and sort within partitions does a repartition and sort by key all in one step, which is more efficient than calling sort by key after a repartition.

Operations and Actions - Part 2

EXAMPLE: CALCULATE AVG PER KEY

- Calculate the average value for each key
- Could do it with groupByKey()

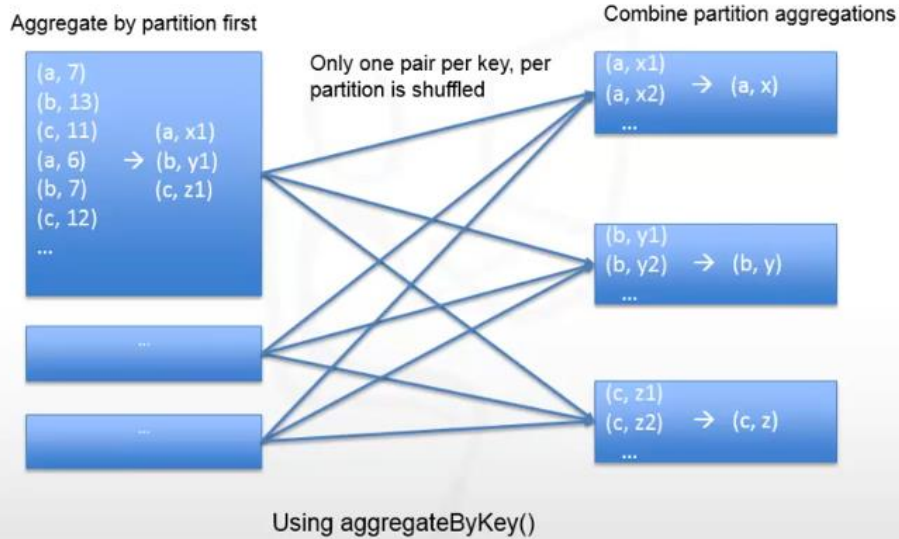
```
val keyedRdd: RDD[(String, Int)] = ???  
keyedRdd.groupByKey().mapValues(l => l.sum / l.size)
```

- **Better:** aggregateByKey()

1. Initial "zero" accumulator

```
val results = keyedRdd.aggregateByKey((0, 0))(  
  (acc, value) => (acc._1 + value, acc._2 + 1),  
  (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2)  
)  
results.mapValues(i => i._1 / i._2)
```

EXAMPLE- CALCULATE AVG PER KEY



EXAMPLE: GET FIRST OF A SEQUENCE

- Get the first value for each key
- Could do it with `groupByKey()`

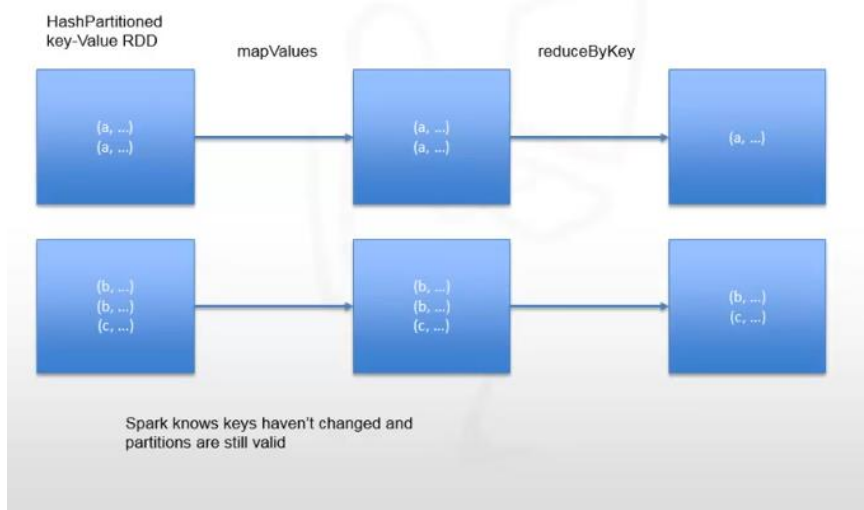
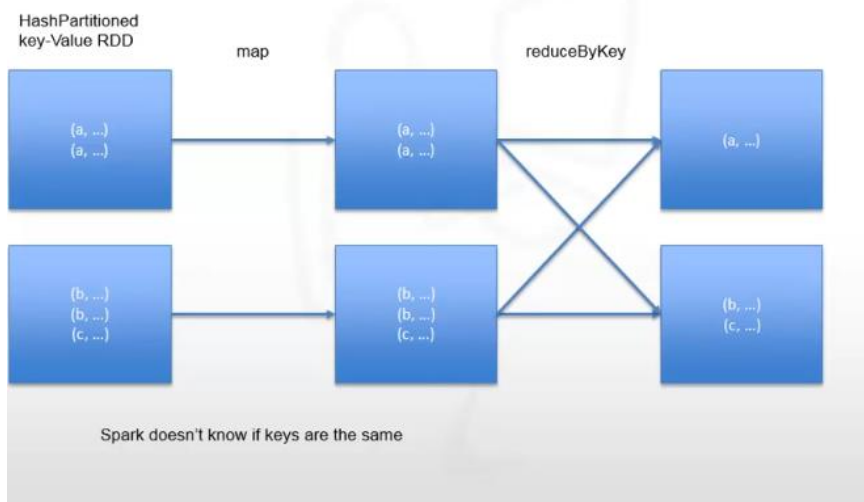
```
val keyedRdd: RDD[(String, Trip)] = ???
keyedRdd.groupByKey().mapValues(l => l.head)
```

- **Better:** `reduceByKey()`

```
val keyedRdd: RDD[(String, Trip)] = ???
keyedRdd.reduceByKey((a, b) => {
  a.startDate < b.startDate match {
    case true => a
    case false => b
  }
})
```


ASYNCHRONOUS ACTIONS

- Actions are blocking operations
- Asynchronous versions of basic actions
- Implemented as futures
 - collectAsync
 - countAsync
 - foreachAsync
 - foreachPartitionAsync
 - taskAsync



WRITING OUTPUT OPERATIONS

- Use saveAsHadoop APIs when possible
- **foreach**: one record at a time
- **foreachPartition**: one partition at a time
- If you need to save to a DB, look at DBOutputFormat
- Use foreachPartition for sending to message queues, REST endpoints, etc.

```
val rdd = ...  
  
rdd.foreachPartition { p =>  
  val conn = initConnection()  
  
  val json_collection = serialize(p)  
  
  conn.post(json_collection)  
}
```

Serializing a whole partition

BROADCAST VARIABLES

- Share a value with every machine
 - Read-only
 - Can be used to avoid shuffling
- Broadcast variable accessible from any RDD
- Shared via peer-to-peer BitTorrent protocol
 - Nodes communicating with each other to optimize throughput
 - Doesn't overload master
- Why not just pass a local variable in closure?
 - Entire closure serialized and sent to every node
 - Delays startup and waste of space

Transformations and Actions - Part 2

BROADCAST EXAMPLE

```

val stations = input2.
  map(_.split(",")).
  map(Station.parse(_)).
  keyBy(_._id).
  collectAsMap()

val sbc = sc.broadcast(stations)

val joined = trips.map(trip => {
  (trip, sbc.value.getOrElse(trip.startTerminal, Nil),
  sbc.value.getOrElse(trip.endTerminal, Nil))
})

```

NO SHUFFLE!

```

(2) MappedRDD[274] at map at <console>:80 □
| MappedRDD[264] at map at <console>:73 □
| MappedRDD[263] at map at <console>:72 □
| FilteredRDD[262] at filter at <console>:71 □
| /Users/silvio/src/adv-spark-labs/data/trips/* MappedRDD[261] at textFile at <console>:62 □
| /Users/silvio/src/adv-spark-labs/data/trips/* HadoopRDD[260] at textFile at <console>:62 □

```

In this example we will see how aggregate functions are used, and also why group by key should be avoided. Consider the following use case: we want to calculate the average value for each key in an RDD. It could easily be done with group by key. Simply call group by key, then find the average for each key by dividing the sum by the number of elements. A better way to do it is with aggregate by key. Let's walk through the code. The initial zero-value accumulator is just 0, 0; a sum of zero for zero total values. The first function iterates all the values for a given key, adding it to the sum and increasing the count of the accumulator for that key by one. Then the second function combines the accumulators for a given key from each partition. Finally, we call mapValues in the same way as before to calculate the average per key. This diagram shows what happens using groupByKey. groupByKey causes a shuffle of ALL values across the network, even if they are already co-located within a partition, then calculates the average per key. In addition to the cost of network I/O you're also using a lot more memory.

In contrast, using aggregateByKey splits the calculation into two steps. Only one pair per key, per partition is shuffled, greatly reducing I/O and memory usage. Then the results from each partition are combined. Another example would be to get the first value of a sequence for each key. Again here you may be tempted to just groupByKey then run a map/mapValues and for each group just grab the head element. The problem once again is the impact to memory this will have with large data sets. A better solution is to use reduceByKey. As each pair of values is passed you can compare them and choose which record to keep. This will run efficiently over all keys in each partition without incurring a large memory overhead. Actions are blocking operations. That is, while transformations are lazily executed, once you call an action such as count, collect, for each, etcetera. your driver has to wait for it to finish. You can get around this with asynchronous operations. These are implemented as futures, so you can set a callback function

to execute once they are completed. The basic actions provide corresponding async versions. You need to be aware that if you plan to executed multiple async operations that if you're using the default FIFO job scheduler they will still complete serially. You'll have to use the FAIR scheduler to compute the actions in parallel.

Understanding the affect of certain operations on partitioning plays a large role in optimizing

transformations. For example, let's say we have a Pair RDD that is already hash partitioned by 2 partitions. If we run a map operation over all records we could potentially transform the keys of each record. Because of that, Spark has no way of knowing if the partitioned keys will match on the other side of the map operation. In this case the resulting RDD will actually no longer have a partitioner. This means even if we don't alter the keys and they remain within the original partitions, when we run a reduceByKey a full shuffle is going to occur. If you just need to operate on the values of each key/pair, always use mapValues. This tells Spark that the hashed keys will remain in their partitions and we can keep the same partitioner across operations. The resulting reduceByKey will now be aware of the partitioning of the keys.

When it comes time to output data from your Spark job you should use the standard save APIs whenever possible. However in some cases you may have custom logic or systems other than HDFS you need to integrate with. It's important to understand the impact of using foreach versus foreachPartition. foreach will run your closure for each record individually. This is obviously not efficient when dealing with large volumes. foreachPartition is more appropriate. Your closure will run in parallel in each partition getting an Iterator for all records within that partition. If you need to save to a standard ODBC or JDBC database consider using the Hadoop DBOutputFormat along with the SparkContext saveAsHadoopDataset

API. You'll need to be careful with the level of parallelism and data volume so that you don't overwhelm your database. For any custom output, such as sending to message queues or REST endpoints then definitely use foreachPartition. Remember to be aware

of any serialization issues. It may be best to just initialize connections within the closure then send everything as one large message

Broadcast variables allow the driver program to send a read-only value to all executors to reference. The broadcast variable can then be looked up by any RDD operations. This can often be used to avoid shuffling. Spark uses a BitTorrent protocol which provides peer-to-peer sharing of the broadcast variable between nodes. This improves network throughput

and reduces the load on the driver. You may be tempted to just pass your data as a local variable closure. The problem here is that you will have to serialize, then transmit the whole serialized closure to each node running that task. This would greatly delay startup and waste space through duplication in every node.

Here is an example using the trips and stations from the lab exercises. Normally, joining the start and end trips to stations would require three shuffles: one to repartition, and one for each join. By broadcasting the stations you can avoid shuffling entirely with some map operations. We'll go over this in detail in the next lab.

Having this lesson, you should be able to understand advanced RDD operations, identify what operations cause shuffling, understand how to avoid shuffling when possible, and work with key value pairs grouping, combining, and reducing. Proceed to exercise 3 and the next lesson.

>>Lab:

Start By Reading The Input Files And Parsing Each Line Into A Trip Or Station Record

```
import java.text.SimpleDateFormat
import java.sql.Timestamp

object utils extends Serializable {
  case class Trip(
    id: Int,
    duration: Int,
    startDate: Timestamp,
    startStation: String,
    startTerminal: Int,
    endDate: Timestamp,
    endStation: String,
    endTerminal: Int,
    bike: Int,
    subscriberType: String,
    zipCode: Option[String]
  )

  object Trip {
    def parse(i: Array[String]) = {
      val fmt = new SimpleDateFormat("M/d/yyyy HH:mm")

      val zip = i.length match { // zip is optional
        case 11 => Some(i(10))
        case _ => None
      }

      Trip(i(0).toInt, i(1).toInt, new Timestamp(fmt.parse(i(2)).getTime), i(3), i(4).toInt, new Timestamp(
        fmt.parse(i(5)).getTime), i(6), i(7).toInt, i(8).toInt, i(9), zip)
    }
  }

  case class Station(
    id: Int,
    name: String,
    lat: Double,
    lon: Double,
    docks: Int,
    landmark: String,
    installDate: Timestamp
  )

  object Station {
    def parse(i: Array[String]) = {
      val fmt = new SimpleDateFormat("M/d/yyyy")

      Station(i(0).toInt, i(1), i(2).toDouble, i(3).toDouble, i(4).toInt, i(5), new Timestamp(fmt.parse(i(6)
        )), getTime))
    }
  }
}

val input1 = sc.textFile("data/trips/*")

val header1 = input1.first // to skip the header row

val trips = ???
```

```
val input1 = sc.textFile("data/trips/*")

val header1 = input1.first // to skip the header row

val trips = input1.
  filter(_ != header1).
  map(_.split(",")).
  map(utils.Trip.parse(_))

val input2 = sc.textFile("data/stations/*")

val header2 = input2.first // to skip the header row

val stations = input2.
  filter(_ != header2).map(_.split(",")).
  map(utils.Trip.parse(_))

val input2 = sc.textFile("data/stations/*")

val header2 = input2.first // to skip the header row

val stations = input2.
  filter(_ != header2).map(_.split(",")).
  map(utils.Station.parse(_))

val byStartTerminal = trips.keyBy(_.startStation)
val durationByStart = byStartTerminal.mapValues(_.duration)
val grouped = durationByStart.groupByKey().mapValues(list => list.sum / list.size)
grouped

import java.text.SimpleDateFormat
import java.sql.Timestamp
defined module utils

//val grouped = durationsByStart.groupByKey().mapValues(list => list.sum / list.size)
//grouped.take(10).foreach(println)

val results = durationsByStart.aggregateByKey((0,0))((acc, value) => (acc._1 + value, acc._2 + 1), (acc1, acc2)
=> (acc1._1 + acc2._2, acc1._2 + acc2._2)

val finalAvg = results.mapValues(i => i._1 / i._2)
finalAvg.take(10).foreach(println)
```

8)

Skip to a navigable version of this video's transcript.

12:46 / 13:07

Maximum Volume.

Skip to end of transcript.

Lab 3

Make sure you open up the lab 3 notebook in Zeppelin.

In this lab you are going to reuse the same bike trips dataset but in this case we're going to have some helper scala classes that will parse each line of the CSV file so that you don't have to work with the raw arrays.

So the objective again for this lab

is to familiarize yourself with the different operation

that there is impact to

job scheduling and performance and you also get to see some advanced pair RDD operations and also to know when to avoid

groupByKey. You've seen some this in the lessons already but now you get a see it in practice. So here's the helper class you can take a moment to understand what it looks like but essentially you're creating a case class for trip and also one for stations and this is just all the column heading of that CSV file We're going to load into here and then create the objects. Here's the trips object You're going to basically parse the data from a CSV file and put it into this object. The same thing is done for the stations. creating the object. the rest of the code here should be familiar to you. You have seen it from the previous lab exercise so now we're going to utilize this helper class to create the trips and the stations RDDs. So for the trips RDD, we're going to first trips we're going to take the input1 and filter out the header as we've done before for each element that doesn't equal the header 1 and then we're going to map it Let's go ahead and format this appropriately so it's easier to see. and then want to map it -- basically split on the commas then you go into map it again this time to that helper class that we just saw and that parse goes through every single value, pretty much. so that will be your trips RDD, and then for the stations we will essentially do the same thing going to first filter it out, so input2, that's where the station's files loaded and filter for each element that doesn't equal the item in the header2 that's the second one. We're going to map it to the commas. split then finally map by the utilities class this should my mistake, it should be stations one object for stations and one for trip. Go ahead and run this panel so that it gets applied Everything here is just a transformation. You shouldn't see any particular output. Expand the output panel. As soon as everything is done here, you can see how you define the module utils and so you defined that up front. Now you want to calculate the average duration by the start terminal using groupByKey and we're going to see how to use groupByKey and later we'll see why you wouldn't want to use groupByKey go ahead and back into the first panel, we're going to first convert this into a key/value pair. The trips and we want only the start terminals because we want to calculate the average

duration of it.

We defined this earlier and now we want to get the duration for each the start terminal by mapping the values which is duration in this case to the key so we define another one finally, group the stations together to get their average duration.

We want to have it print out or invoke an action.

go ahead and run this panel

and then we defined the startTerminal

map to durations

mapping the startTerminal to the duration

grouping it together to do the

average durations so this

station name and this is the average duration

first 10 from the list and then we print it out. So the next thing we want to do

using a groupByKey but how do you think we should optimize this?

and you saw this in the lesson already. The one reason

when you're using a groupByKey, the key/value pairs are all shuffled around

essentially each of the partitions, they just send all the key

send out based on the keys to the final partition

and when they do that and it ends up using a lot of memory when the dataset is really large

you are just essentially shooting out keys everywhere right.

so the alternative here to make it more efficient

and prevent lot of shuffling is to use the aggregateByKey

so what happens there is aggregateByKey will combine the keys

first within its own partition before

sending it out so for example you have 10 keys

in your node, in your

worker node, so 5 of those 10

records have the same key, so let's combine those 5 together

into one so instead emitting all 10 records out

you just emitting 6, so you combined the 5 into 1

and then there's the other 5 that are different so and in that small case

it probably doesn't make much of a difference, 6 vs 10

but assuming you like tons and tons of data, as we're dealing with here, that will

make a lot of difference. So how we do that?

Let's go up to our panel.

and now instead of the groupByKey

we're going to do the aggregateByKey.

Let's comment out the groupByKey

portion. We're not going to run this.

and then use the aggregateByKey

method.

and we're starting with zero, zero in this

accumulator as well as to the actual value

so one is for the count and the other one is for the sum.

the getting the first value

index1 adding it to the value and then

getting the second value. Both of these are zero to begin with, but later they will accumulate

as

more values are brought into the mix and then the other one is going to add the value of one -- that's for the count. The second function here is going to take that first one. This is basically the function that will be used or invoked, or called upon once everything is combined at the final worker node. and next thing you want to do is compute average so you have the result now you can compute the average sum of everything and the second one is the count of everything. That's how you calculate the average alright looks good now great. If you paid attention to the time it takes on this one should have completed a little bit quicker, but again, you know, unless you are dealing with an extremely large dataset time this negligible. However it does make a big difference So that is what I wanted to point out by using the `aggregateByKey` will reduce a lot of unnecessary shuffling doing a lot of the computations up front and however remember that not all `groupByKey` use cases can be replaced with the `aggregateByKey` or even the `reduceByKey` operations you have to know that only a certain use case like this one here when you are calculating the average you can do the aggregation initially first, before combining it but let's say the order or something matters, then you have to wait until all the keys are group together before you can do the operation. In those cases `groupByKey` still works and it may not be efficient but that is the way for those types of use cases.

MODULE 4 CACHING AND SERIALIZATION

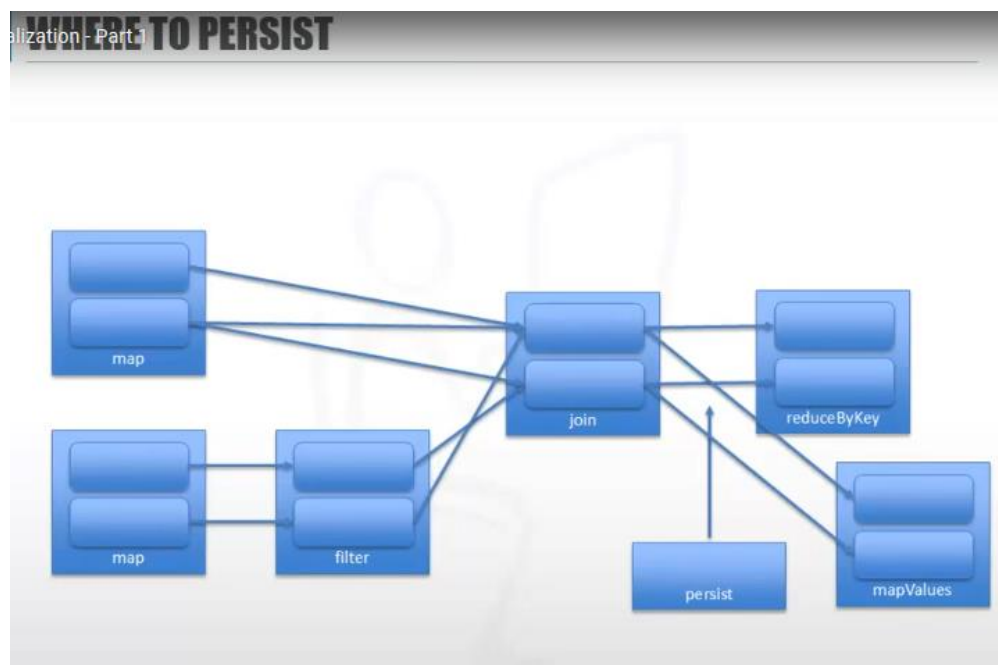
PERSISTENCE

- Spark's power comes from memory caching
 - Minimal disk I/O
- Important to know when to **persist** RDDs
- Different types of persistence have tradeoffs
 - Memory/disk combination for large data
 - Save to disk to avoid losing results of expensive operation
- Good idea to persist after filter, other prep for downstream processing
- Call `unpersist()` when no longer needed
 - Spark uses LRU algorithm to make room when needed

```
val input1 = sc.textFile("data/trips/*.csv")

val trips = input1
  .filter(!_._1.startsWith("Trip ID")).map(_._2.split(",")).
  .map(Trip.parse(_)).
  .keyBy(_._1.startTerminal).
  .partitionBy(new HashPartitioner(4))
```

Good place to persist



STORAGE LEVELS

cache() == persist(StorageLevel.MEMORY_ONLY)

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as <i>serialized</i> Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a <i>fast serializer</i> , but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store RDD in serialized format in <i>Tachyon</i> . Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.

STORAGE COST

- Persistence uses memory
- Raw Java objects are fast but take up most space
- Instead, we can serialize
 - Less space at cost of CPU
 - Stored as byte array
 - Also helps with garbage collection (1 array vs many objects)
- Defaults to **Java** serialization (Python uses **pickle**)
- **Kryo** is more efficient, but more configuration involved
- Compress to save further space at CPU cost of decompressing
- Primitive types store better than Java/Scala collections and heavily nested classes

Lesson 4: Caching and Serialization

This course was developed in collaboration with MetiStream and IBM Analytics.

After completing this lesson you should be able to understand how and when to cache RDDs, understand storage levels and their uses, optimize memory usage with serialization options, and share RDDs with Tachyon.

Spark is very well suited to iterative analysis because of how it distributes and executes tasks across the cluster and the fact that it minimizes disk I/O compared to frameworks such as MapReduce. However, sometimes we may need some additional performance gains while

dealing with the same RDD over numerous iterations. What Spark is most known for is its use of memory caching. It's important to understand when and how we should persist RDDs.

Persistence

doesn't necessarily have to be in-memory only. We can use a combination if we know the data is too large to fit in memory or we could choose to save entirely to disk if we're more concerned with losing results of an expensive operation. Persisting to disk would allow us to reconstitute the RDD from disk in the event a partition is lost, instead of re-computing all the expensive operations for the lost partitions.

Ideally you want to persist after any pruning, filtering, and other transformations needed for downstream processing. An example is loading a file, parsing it, and partitioning it by a key. It wouldn't do much good to simply cache the root RDD if we would have to incur the filter, map, and shuffle again each time we re-use that RDD. Instead, we'd be better off caching after the partitionBy. When you no longer need the persisted RDD simply call un-persist. Spark will also use a Least Recently Used (LRU) algorithm as needed to make room for new RDDs. In this RDD lineage we're joining 2 RDDs, one which is filtered. Then we have two branching operations: a reduceByKey and a mapValues.

It would be a good idea to persist after the join, when the RDD is prepped and ready for the reduce and map actions downstream. These are the different storage levels available, from the Apache Spark programming guide. MEMORY_ONLY is the default. You can call the shorthand

cache function to persist with the default. Note how the different levels handle overflow. With memory only, if the whole RDD can't fit in memory, some of the partitions will have to be recomputed on the fly. The DISK options write the overflow to disk instead.

You can also replicate persisted partitions, or store them in Tachyon, a way to "share" RDDs that we will talk about shortly. While persisting RDDs might help us save time re-computing partitions it does come at a cost. That is the space it takes to keep all those objects in memory. Keeping records in memory in a raw state takes the most space. To help save on space we can serialize. The records of an RDD will be stored as one large byte array. You'll incur some CPU usage to deserialize the data. However it has the added benefit of helping with garbage collection as you'll be storing 1 object (the byte array) versus many small objects for each record.

The default serializer is the Java serializer, or pickle if you're using python.

Another option for Java and Scala users is the Kryo serializer, which is more efficient at the cost of some potential compatibility or configuration issues.

You can also compress serialized in-memory RDDs at the cost of decompressing

As a rule of thumb optimize storage by using primitive types instead of Java or Scala collections.

Also try to avoid nested classes with many small objects as that will incur large garbage collection overhead.

RDD from 36MB CSV (Java)

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
22	Memory Deserialized 1x Replicated	2	100%	162.8 MB	0.0 B	0.0 B
52	Memory Serialized 1x Replicated	2	100%	52.3 MB	0.0 B	0.0 B
27	Memory Deserialized 1x Replicated	2	100%	19.6 KB	0.0 B	0.0 B
57	Memory Serialized 1x Replicated	2	100%	10.8 KB	0.0 B	0.0 B


```
val conf = new SparkConf().setAppName("lab4")
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[Trip], classOf[Station]))

val sc = new SparkContext(conf)

val data: RDD[Trip] = _

data.persist(StorageLevel.MEMORY_ONLY_SER)
```

Simple Kryo configuration

CAN WE "SHARE" RDDS?

- Not conventionally
 - RDD tied to SparkContext
- Tachyon project - <http://tachyon-project.org/>
- Off-heap cache for Spark
- Periodically save to or read from Tachyon
- Can persist RDDs for other Spark apps to use

```
rdd.persist(StorageLevel.OFF_HEAP)
```



MEMORY CONFIGURATION

Some key configuration parameters to understand:

- `spark.rdd.compress` (default=false)
- `spark.shuffle consolidateFiles` (default=false)
 - Set *true* on ext4 or xfs file systems
 - May degrade performance on ext3 with > 8 cores
- `spark.shuffle.spill` (default=true)
 - Limit memory used during reduces by spilling to disk
 - `spark.shuffle.memoryFraction` (default=0.2)
 - `spark.shuffle.spill.compress` (default=true)
- `spark.storage.memoryFraction` (default=0.6)
 - How much storage can be dedicated to in-memory persistence
- `spark.storage.unrollFraction` (default=0.2)
 - memory dedicated to “unrolling” serialized data

Here’s an example of the trips and stations RDDs you’ve been using in the labs, cached both with and without serialization. While the uncompressed input file was roughly 36MB, the persisted dataset is much larger because of the size of the Java objects in memory. In contrast the serialized version, using the default Java serializer, only uses 52MB of space. The Kryo serializer actually manages to store the entire RDD in less space than the original file!

Kryo is usually very straightforward to set up. Here we see how to set up Kryo for the lab exercises. First you create a new `SparkConf` configuration for the app. Then you set the spark serializer parameter to `KryoSerializer`. You must then register the classes in your application that will be serialized, in this case, the `Trip` and `Station` classes. Finally you initialize spark context with the configuration. Now when you use a persist level such as `MEMORY_ONLY_SER` the serialization will be handled by Kryo.

One common question when people first start working with Spark is if RDDs can be shared across applications. Since RDDs are tied to a `SparkContext` this is not conventionally possible. Cue the Tachyon project, a high-speed memory-based Distributed File System that comes with Spark. You can use Tachyon to save and read RDDs across `SparkContexts` in a very

efficient manner. The experimental `OFF_HEAP` storage level can also be used to persist RDDs just as you would on disk or in RAM of the Spark cluster.

There are some key configuration parameters you should be familiar with.

`spark.rdd.compress` decides whether or not serialized RDDs should also be compressed.

`spark.shuffle consolidateFiles` consolidates the intermediate files generated during a shuffle, creating fewer larger files rather than many small ones potentially improving disk I/O. Docs suggest setting this to true on ext4 or xfs filesystems. On ext3 this might degrade performance on machines with more than 8 cores.

`spark.shuffle.spill` limits the amount of memory used during reduces by spilling data out to disk. The spilling threshold is specified by `spark.shuffle.memoryFraction`. You can also

Spark II (Cognitive Class)

set whether or not the spill will be compressed. `spark.storage.memoryFraction` is how much storage

can be dedicated to in-memory persistence. `spark.storage.unrollFraction` is memory dedicated

to “unrolling” serialized data such as persisted RDDs since they’re stored as one large byte-array.

Having completed this lesson you should be able to

Understand how and when to cache RDDs,

understand storage levels and their uses,

optimize memory usage with serialization options, and share RDDs with Tachyon.

Proceed to exercise 4 and the next lesson.

>>Lab:

1. First run the job without any caching
2. Now try calling `cache` on the `trips` RDD and re-run the transformation
3. Finally, use `persist(StorageLevel.MEMORY_ONLY_SER)` and re-run, compare the storage statistics in the Spark UI

Took 0 seconds

Solution

FINISHED ▶ ⌂ ⚙

```
val durationsByStart = trips.keyBy(_.startTerminal).mapValues(_.duration)
val durationsByEnd = trips.keyBy(_.endTerminal).mapValues(_.duration)

val resultsStart = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 + value, acc._2 + 1),
  (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avgStart = resultsStart.mapValues(i => i._1 / i._2)

val resultsEnd = durationsByEnd.aggregateByKey((0, 0))((acc, value) => (acc._1 + value, acc._2 + 1), (acc1,
  acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avgEnd = resultsEnd.mapValues(i => i._1 / i._2)

avgStart.collect()
avgEnd.collect()
```

Using The Fastutil Collection Library For Optimized Data Structures

FINISHED

```
import it.unimi.dsi.fastutil.objects.Reference2ReferenceArrayMap
import org.apache.spark.storage.StorageLevel

// the rest of the code is the same

val rdd = sc.parallelize(1 to 10) map { i =>
  | val pageDwellTimes = new Reference2ReferenceArrayMap[String, Int]() // fastutil version
  | | val pageDwellTimes = scala.collection.mutable.Map[String, Int]() // standard Scala collection
  |
  | 1 to 400 foreach { i => pageDwellTimes.put(s"page $i", 30) }
  |
  | (s"user $i", pageDwellTimes)
  | }
rdd.count()
```

In lab 4

we're going to cover caching and memory tuning and you're actually using the same `trips` and `stations` RDD we had used previously

so let's take a look here

same helper class. Everything has been done already. So what you want to do is

now we're going to calculate the average duration

of trips by both the start and the end terminals. You can refer back to lab 3 for the transformations to run the calculations or

quicker away with just to be copy the solution
I won't blame you if you copied it here instead of typing.
Remember you need to copy the
two actions too.
If I didn't have the action, you wouldn't see any difference
because nothing would have been executed. So the first task
is to run a job without any caching just to set the benchmark
in a way. I went ahead and ran this
Expand the output. Go back down.
Alright, so we did that,
now let's cache it
Caching the trips. Why do we want to do that? Well, let's see here. You notice that the
trips
is being used here and then the trips is being used here.
In spark,
when you use the same RDD in multiple locations and if it wasn't cached,
what happens is it actually re-computes the
trips RDD follow down the lineage from the root. So actually every time you call it
it's going to go through
this one and then all the way back to re-loading the file.
It has to redo everything over. That's just the way Spark is
designed to do, but when you cache it it would just take
the latest level or the latest state
of that RDD from when it was cached and just reuse that.
Here we're caching by memory only.
In fact this is pretty much the same thing as if you were to call trips.cache
In fact, cache calls persist with storage level,
MEMORY_ONLY, so they're the same thing, but we're just going to call it like that and
go ahead and run it with this
MEMORY_ONLY caching, which is the default.
Now, let's do the same thing, but this time we want to serialize it as well.
So run with the MEMORY_ONLY_SER
ok -- they both completed. You won't really see much of a difference here,
but you can check out the Spark UI which is on the port 4040 and I
open it up here.
Under the storage tab, did a quick refresh to make sure
it loaded. Essentially, it is on port
4040. You can see the two levels here.
The serialized version, much
less memory usage and the
deserialized version uses almost three times as much in this case.
Let's go back.
This is basic default serialization using the java serializer
We are going to use the Kryo serialization.
That's actually the second part of lab 4, and I wanted to get into that part first.
Since we're were just talking about it now. Go ahead and click the notebook.
We will continue with the last part of lab 4
after, but I want to get into the Kryo
serialization
So here we are again, same thing, but
this Zeppelin here is configured to use the
Kryo serialization. In this case you have to register your classes

so that's one of the main differences between Kryo and Java
In Kryo you have to explicitly register your classes to use it
but it is much quicker and more efficient than Java.
So we're going to do the same thing, I'm going to grab the solutions here
paste
everything in that first panel of ours and let's run it with
the storage MEMORY_ONLY
now this is going to run
with the Kryo serialization. Actually, that one is just the default cache.
So there's nothing different yet.
Let's hide this one and uncomment this one out.
now we are going to serialize using the Kryo serialization.
run it
ok -- looks like it completed. We had configured the
docker container to use a different port
for the Spark UI. It's actually on port 4041.
so I'm copying it over, 4041.
So the deserialized is the roughly the same, obviously it's the same dataset
but notice in the serialized version
using lot less space than the
Java one, so basically
Kryo serialization is more efficient and more optimized
for it so that's the recommended one you would use.
instead of the Java one, but remember, you have to register the
classes, so, let me show you what that looks like.
This is saying that we open it on port
4041 and then you need to configure the
serialization for the Kryo library
so on the classes here, I want to show you at the very bottom
where you have to explicitly register these classes so we register
trip class and register the station class.
Remember these classes were created by our helper
classes we define previously.
Let's jump back over
to lab 4 and finish up this final task
at the bottom here. This is just to introduce you to the fastutil
library collection. There's nothing you have to do here
It's just to introduce you to this code, and it uses the
standard scala collection, and run it.
Then, you can run the same code again using the fastutil
version, which is more efficient in terms of smaller memory footprint
and you can again just run it like this
see the output and you see the minor time differences,
and obviously, if you wish to
learn more about this library, you can visit the -- just google for the fastutil
library, and you will be able find and learn more about this library.
It's just something that's more efficient than what you normally would do in
Java
for memory tuning.
That's pretty much it here. This pretty much concludes lab 4.
You should kind of understand how to use cache and
be familiar with the different persist and the storage levels

But it is often good practice to unpersist the cache once you are done with it. That way you can save additional memory, so you are no longer using and remember to call the unpersist method on the RDD in order to un cache the data from memory and the difference between Kryo and the Java serialization. Kryo is usually more efficient in most cases than Java but the tradeoff is you would need to explicitly register the classes. So this concludes lab 4.

MODULE 5 DEVELOP AND TESTING

esting

BUILD TOOLS

- sbt
 - Built-in REPL
 - Build and run from terminal
 - Simple, but powerful and extensible
 - Plugins for Eclipse and support in IntelliJ
- Maven
 - Not as powerful or customizable as sbt

```
name := "adv-spark-labs"
version := "0.0.1"
scalaVersion := "2.10.4"
val sparkVersion = "1.2.1"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion
)
```

sbt.build file

CREATE A PROJECT FROM COMMAND LINE

```
Loading /usr/local/sbt-0.13/bin/sbt-launch-lib.bash
[info] Set current project to src (in build file:/Users/silvio/src/)
> set name := "my-project"
[info] Defining *:name
[info] The new value will be used by *:description, *:normalizedName and 6 others.
[info] Run 'last' for details.
[info] Reapplying settings...

$ mkdir hello
$ cd hello
$ echo 'object Hi { def main(args: Array[String]) = println("Hi!") }' > hw.scala
$ sbt
...
> run
...
Hi!
```

Source: www.scala-sbt.org/0.13/tutorial/Hello.html

SBT AND ECLIPSE

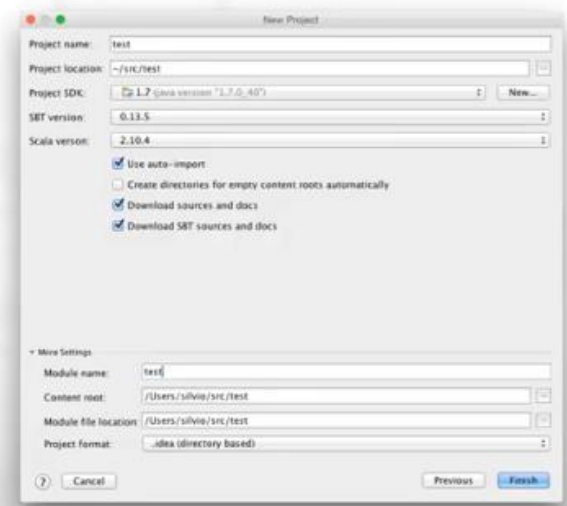
- sbt-eclipse plugin to generate Eclipse project files
- Add the following to project/plugins.sbt file:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "3.0.0")
```
- From terminal in root directory of project:

```
> sbt eclipse
```
- Import existing Eclipse project
- Need to re-run any time build changes
- <https://github.com/typesafehub/sbteclipse>

SBT AND INTELLIJ

- Much cleaner sbt integration with plugin
- Scala console built-in
- Supports scalatest
- Debug support using remote debugger



UNIT TESTING YOUR SPARK APPS

- Isolate your RDD operations
 - Encapsulate logic in its own object/class
 - Same code is run during test as in application
- Use standard unit testing tools, like scalatest
- Spark Packages – spark-testing-base
 - Helper classes to facilitate testing Spark apps

```
libraryDependencies ++= Seq(  
  "org.apache.spark" %% "spark-core" % sparkVersion % "provided",  
  "com.holdenkarau" %% "spark-testing-base" % "1.2.0_1.1.1_0.0.6" % "test",  
  "org.scalatest" %% "scalatest" % "2.2.1" % "test"  
)
```

sbt testing dependencies

UNIT TESTING EXAMPLE

```
class SampleTests extends FunSuite with SharedSparkContext {

  test("word count") {
    val input = "hello hello world"
    val expected = Map("hello" -> 2, "world" -> 1)

    val counts = sc.makeRDD(Seq(input)).
      flatMap(_ split(" ")).
      map(_ 1).
      reduceByKey(_ + _).
      collectAsMap

    assertResult(expected) {
      counts
    }
  }

  test("line parser") {
    val input = "helloworld"
    val expected = Seq("hello", "world")
    val result = input.split(" ")

    assertResult(expected) {
      result
    }
  }
}
```

Should be in its own class/object

MANAGING DEPENDENCIES

- Additional libraries must be distributed to workers
- With spark-submit, use --jars flag
 - Can reference http, ftp, hdfs
- Bundle everything into one big “fat jar”
- Easy in sbt
 - addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
 - run sbt assembly

Lesson 5: Developing and Testing

This course was developed in collaboration with MetiStream and IBM Analytics. After completing this lesson, you should be able to use sbt to build Spark projects, use Eclipse and IntelliJ for Spark development, unit test your spark projects, and manage dependencies.

Sbt is the most popular built tool for Scala. It has built in REPL interface that allows you to build and run from the shell. It's simple to use but powerful and extensible.

It has plugins for Eclipse and direct support in the IntelliJ IDE

If you're already using it or just prefer the familiarity you can also use Maven, but

it's not as powerful or customizable. Here's a basic sbt build file.

With SBT you can also create builds directly from the console. For example, renaming your current project. In fact, sbt can build and run without any configuration files at all. It will automatically find source and library files using a conventional directory structure. Visit the sbt project page and the tutorials there for more information.

You can use the sbt Eclipse plugin to generate the files for an Eclipse project.

Just add this line to your project's plugins.sbt file.

Then from the terminal, in the root directory of your project, run `sbt eclipse`.

Now you can import the project from within Eclipse.

This needs to be re-run every time the build changes.

View the github page for the plugin for more information.

IntelliJ has much cleaner integration with sbt.

It fully supports sbt build files with no conversions required.

IntelliJ has a built-in Scala console, it supports scala test and debugging.

We'll go over a few points to get you started with unit testing.

Isolate your RDD operations. The transformations for a given RDD should be put in its own object

or class. You want to test the code that is actually

used in your application. Take advantage of standard unit testing tools,

like scalatest. There is also the very handy spark-testing-base package that comes with helper class to facilitate testing. Here is part of an sbt build that includes

scalatest and spark-testing-base. Here is an example unit test of a word count.

We have our test input and the expected results. Then we run a word count on the input.

Realistically

the logic for the word count would be in a class or object somewhere else. Then we run

`assertResults` to check that the expected value matches the output.

Here's another test. Again, this is just a contrived example for demonstration purposes.

`assertResult` tests that splitting the input gives the expected sequence. But in this case the test will fail, since our input string doesn't have any spaces in it.

A `TestFailedException` will be thrown, showing the expected and actual results of the test.

Any additional libraries in your project must be distributed to workers.

When using the spark-submit script, use the jars flag. You can reference jars from many sources including HTTP, FTP, and HDFS. Common practice is to bundle everything into one big jar. This is made easy with sbt. just add sbt-assembly

to your plugins file. Then run `sbt assembly` in your project root directory.

Having completed this lesson, you should be able to use sbt to build Spark projects, use Eclipse and IntelliJ for Spark development, unit test your spark projects, and manage dependencies.

Congratulations on completing the course.

Don't forget to do the final exercise and the course quiz, and then fill out the course feedback. Thank you.

>>Lab:

Creating A Basic Sbt Project

1. First start a command line and create a directory for your project, call it lab5
2. Inside the new directory, run the sbt command
3. Once the sbt console comes up, enter the following commands:

```
set name := "lab5"
set version := "1.0"
set scalaVersion := "2.10.4"
session save
exit
```

4. Once you exit, you should have a basic build.sbt file along with a project directory
5. Edit the build.sbt file to verify its settings match what you entered above
6. Add the following lines to the build.sbt file:

```
val sparkVersion = "1.2.1"

libraryDependencies += Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion
)
```

7. Now create the following directory structure:

```
src/
  main/
    resources/
    scala/
  test/
    resources/
    scala/
```

http://192.16...oteboo
192.168.59.103:8080

Zeppelin

setAppName
val trips =
val station
trips.count
stations.co
println("Pr
Console.rea
sc.stop
}
}

```
[info] Run 'last' for details.
[info] Reapplying settings...
[info] Set current project to lab5 (in build file:/C:/lab5/)
> session save
[info] Reapplying settings...
[info] Set current project to lab5 (in build file:/C:/lab5/)
> exit

C:\lab5>dir
Volume in drive C has no label.
Volume Serial Number is 1A85-9E7E

Directory of C:\lab5
06/04/2015  05:36 PM  <DIR>          .
06/04/2015  05:36 PM  <DIR>          ..
06/04/2015  05:36 PM               66 build.sbt
06/04/2015  05:35 PM  <DIR>          project
06/04/2015  05:35 PM  <DIR>          target
                        1 File(s)          66 bytes
                        4 Dir(s)  75,837,276,160 bytes free

C:\lab5>notepad build.sbt

C:\lab5>
```

10. From the root of the lab5 project, run:

sbt compile

11. Verify no errors occurred

12. Now let's create a JAR we can submit to Spark

sbt package

13. The JAR should be in `target/scala-2.10/lab5_2.10-1.0.jar`

Took 0 seconds

8. Create a file Lab5.scala under src/main/scala

9. Copy this code below into the file:

```
package lab5

import org.apache.spark.{SparkConf, SparkContext}

object Lab5 {

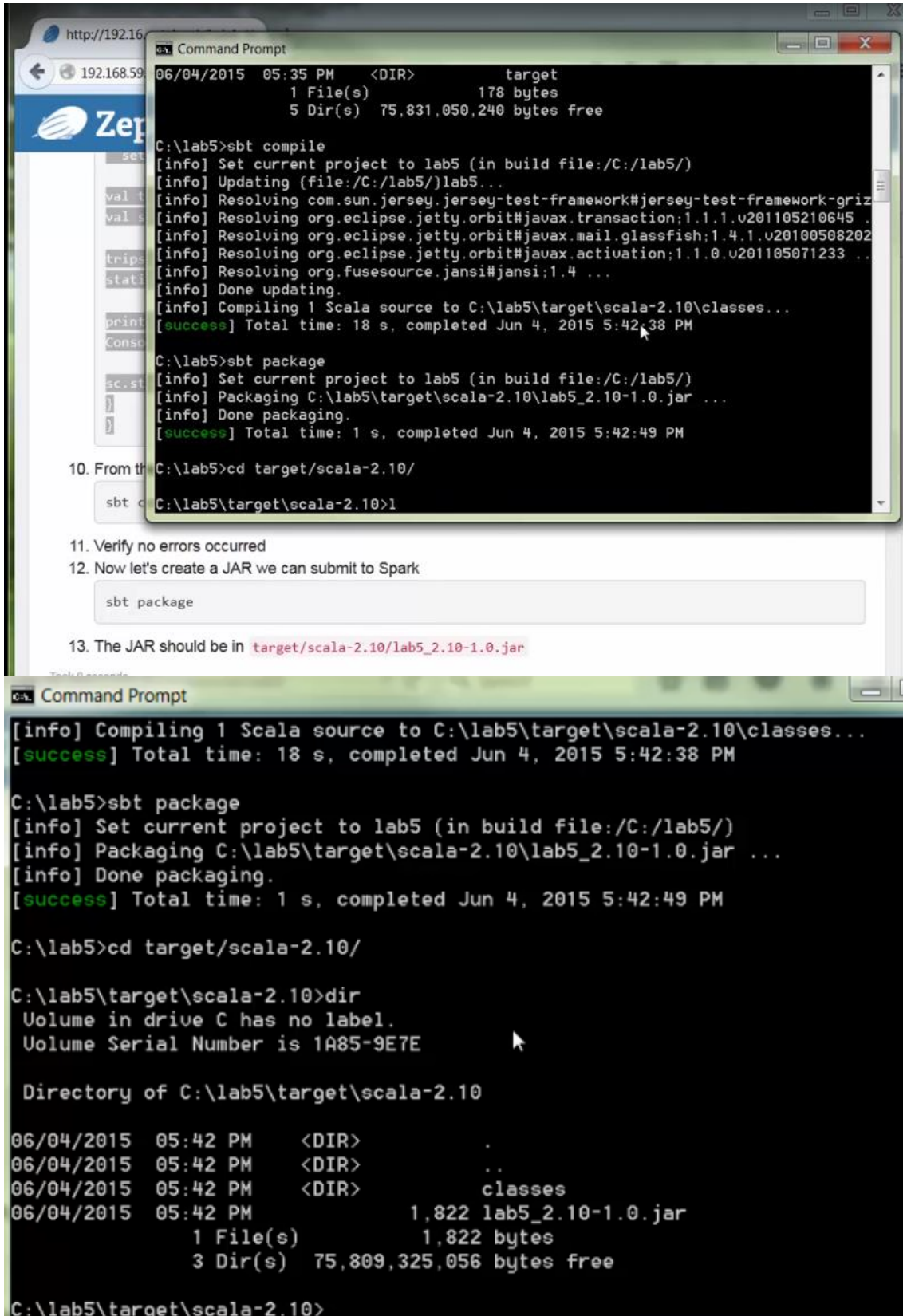
  def main(args: Array[String]) = {
    val sc = new SparkContext(new SparkConf().
      setAppName("lab5"))

    val trips = sc.textFile("data/trips/*")
    val stations = sc.textFile("data/stations/*")

    trips.count()
    stations.count()

    println("Press [Enter] to quit")
    Console.readLine()

    sc.stop
  }
}
```

The screenshot shows a web browser window with a URL bar displaying 'http://192.168.59...' and a 'Zep' logo. Overlaid on the browser is a Windows Command Prompt window. The Command Prompt shows the following commands and output:

```

C:\lab5>dir target
06/04/2015  05:35 PM    <DIR>          target
               1 File(s)            178 bytes
               5 Dir(s)  75,831,050,240 bytes free

C:\lab5>sbt compile
[info] Set current project to lab5 (in build file:/C:/lab5/)
[info] Updating {file:/C:/lab5/}lab5...
[info] Resolving com.sun.jersey.jersey-test-framework#jersey-test-framework-grizzly2:1.17.1...
[info] Resolving org.eclipse.jetty.orbit#javax.transaction:1.1.1.v201105210645...
[info] Resolving org.eclipse.jetty.orbit#javax.mail.glassfish:1.4.1.v201005082022...
[info] Resolving org.eclipse.jetty.orbit#javax.activation:1.1.0.v201105071233...
[info] Resolving org.fusesource.jansi#jansi:1.4...
[info] Done updating.
[info] Compiling 1 Scala source to C:\lab5\target\scala-2.10\classes...
[success] Total time: 18 s, completed Jun 4, 2015 5:42:38 PM

C:\lab5>sbt package
[info] Set current project to lab5 (in build file:/C:/lab5/)
[info] Packaging C:\lab5\target\scala-2.10\lab5_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 1 s, completed Jun 4, 2015 5:42:49 PM

10. From the Command Prompt, run:
C:\lab5>cd target/scala-2.10/
C:\lab5\target\scala-2.10>ls

11. Verify no errors occurred
12. Now let's create a JAR we can submit to Spark
C:\lab5>sbt package

13. The JAR should be in target/scala-2.10/lab5_2.10-1.0.jar
  
```

Below the Command Prompt window, the text 'Task 0 completed' is visible. Below that, another Command Prompt window is shown, displaying the following commands and output:

```

C:\lab5>sbt compile
[info] Compiling 1 Scala source to C:\lab5\target\scala-2.10\classes...
[success] Total time: 18 s, completed Jun 4, 2015 5:42:38 PM

C:\lab5>sbt package
[info] Set current project to lab5 (in build file:/C:/lab5/)
[info] Packaging C:\lab5\target\scala-2.10\lab5_2.10-1.0.jar ...
[info] Done packaging.
[success] Total time: 1 s, completed Jun 4, 2015 5:42:49 PM

C:\lab5>cd target/scala-2.10/

C:\lab5\target\scala-2.10>dir
Volume in drive C has no label.
Volume Serial Number is 1A85-9E7E

Directory of C:\lab5\target\scala-2.10

06/04/2015  05:42 PM    <DIR>          .
06/04/2015  05:42 PM    <DIR>          ..
06/04/2015  05:42 PM    <DIR>          classes
06/04/2015  05:42 PM                1,822 lab5_2.10-1.0.jar
               1 File(s)            1,822 bytes
               3 Dir(s)  75,809,325,056 bytes free

C:\lab5\target\scala-2.10>
  
```

Using Eclipse

FINISHED ▶ 🔍 📄

To use Eclipse with an sbt project, we first have to enable the eclipse-sbt plugin. This plugin will generate an Eclipse project for us that we can then import. It doesn't provide all the advanced functionality we get with sbt but provides a familiar environment if you prefer Eclipse.

1. Create a file plugins.sbt in the "project" folder
2. Add the following line to the plugins.sbt file:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "3.0.0")
```

3. Run the eclipse task from the sbt command line:

```
sbt eclipse
```

4. Open Eclipse and import a new project, using the root directory of our app
5. Verify that all dependencies and files are included in the project and that it compiles properly

Took 0 seconds

First install the Scala plugin

1. Start IntelliJ
2. Go to Preferences -> Plugins
3. Search for "Scala" and enable the plugin

Now open an existing project

1. Go to open a project
2. Browse to the path of our Lab5 app and open it
3. Select "Use auto-import" and "Create directories for empty content roots automatically"
4. Set the 'Project SDK' to 1.7
5. Click 'OK'

Now let's try creating a new project from scratch

1. Select "Create New Project"
2. Make sure Scala is selected and SBT as the project type
3. Enter a project name, location (different than the one above)
4. Set SDK to 1.7, SBT to the latest 0.13., *Scala to the latest 2.10.*
5. Check "Use auto-import" and "Create directories for empty content roots automatically"
6. Uncheck "Download sources and docs" and "Download SBT sources and docs"
7. Click finish
8. Open the build.sbt file and paste the contents below:

```
val sparkVersion = "1.2.1"
```

Debugging In Eclipse

FINISHED ▶ 🔍 📄

For this assignment, we'll use the project you have already loaded in Eclipse and sbt

1. In Eclipse create a new Debug Configuration for a Scala Application
2. Set a VM argument such as: `-Dspark.master=local[*]`
3. Set a breakpoint on the first line of the Lab5 app
4. Start the debugger using the Debug Configuration you created
5. You should shortly hit your breakpoint in Eclipse

Took 0 seconds

Debugging In IntelliJ

FINISHED ▶ ⚙️ 📄 🔍

For this assignment, we'll use one of the previous projects you have running in IntelliJ and sbt.

1. In IntelliJ create a new Run Configuration for an Application
2. In the VM Options enter `-Dspark.master=local[*]`
3. Set a breakpoint on the first line of the Lab5 app
4. Start the debugger using the Run Configuration you created
5. You should shortly hit your breakpoint in IntelliJ

Took 0 seconds

1. Add code to skip the header in each CSV file
2. Use the code below to calculate the distance between two latitude/longitude points (note: this is simply calculating the direct distance)

```
def distanceOf(lat1: Double, lon1: Double, lat2: Double, lon2: Double): Double = {
  val earthRadius = 3963 - 13 * Math.sin(lat1)

  val dLat = Math.toRadians(lat2-lat1)
  val dLon = Math.toRadians(lon2-lon1)

  val a = Math.pow(Math.sin(dLat / 2), 2) + Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) * Math.pow(Math.sin(dLon / 2), 2)

  val c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a))

  val dist = earthRadius * c

  dist
}
```



```
case class Trip(
  id: Int,
  duration: Int,
  startDate: String,
  startStation: String,
  startTerminal: Int,
  endDate: String,
  endStation: String,
  endTerminal: Int,
  bike: Int,
  subscriberType: String,
  zipCode: Option[String]
)

object Trip {
  def parse(i: Array[String]) = {
    val zip = i.length match {
```

```
        name: String,  
        lat: Double,  
        lon: Double,  
        docks: Int,  
        landmark: String,  
        installDate: String  
    )  
  
    object Station {  
        def parse(i: Array[String]) = {  
            Station(i(0).toInt, i(1), i(2).toDouble, i(3).toDouble, i(4).toInt, i(5), i(6))  
        }  
  
        val default = Station(0, "None", 0, 0, 0, "", "")  
    }
```

4. Write a transformation that calculates the distance of each trip and outputs the trip id and distance values
5. Save the data out to HDFS
6. Build your JAR using your IDE or the command line

```
sbt package
```